Computer Science 460/660
Introduction to Database Systems

Boston University, Fall 2013
David G. Sullivan, Ph.D.

Syllabus................................................................................................................ 1
Introduction and Course Overview...................................................................... 5
Database Design and the Entity-Relational Model .............................................. 17
The Relational Model and Relational Algebra..................................................... 36
The SQL Query Language ................................................................................ 63
Storage and Indexing ........................................................................................ 92
Implementing a Logical-to-Physical Mapping .................................................... 117
Query Processing ............................................................................................. 128
Transactions .................................................................................................... 142
Concurrency Control ....................................................................................... 155
Recovery and Logging ..................................................................................... 180
Objects in Database Systems .......................................................................... 202
Semistructured Data and XML ....................................................................... 222
Distributed Databases and Replication.............................................................. 253
Performance Tuning........................................................................................ 273

Notes for additional topics will be distributed in lecture.
Description: This course covers the fundamental concepts of database systems. Topics include data models (ER, relational, and others); query languages (relational algebra, SQL, and others); implementation techniques of database management systems (indexing structures, concurrency control, recovery, and query processing); management of semistructured and complex data; and distributed databases.

Prerequisites: CAS CS 112, or equivalent

Instructor
David G. Sullivan, Ph.D. (dgs @ cs . bu . edu, removing the spaces)
office hours: Mon., 3-5 p.m.; Wed., 2-4 p.m., and by appointment
office: Psychology Building (PSY), room 228D
64 Cummington Mall (behind Warren Towers)

Teaching Fellow
Haohan Zhu (zhu @ cs . bu . edu, removing the spaces)
office hours: TBA

Lectures: Tues/Thurs, 9:30-11:00 a.m., KCB 106

Course Website: http://www.cs.bu.edu/courses/cs460

Requirements
1. Five problem sets, including a combination of written exercises and programming problems.
2. Midterm exam
3. Final exam
4. Attendance at and participation in the lectures
CS 660 students will complete additional work.
The exams are closed-book.

Policies
Lateness policy: Assignments must be submitted by the date and time listed on the assignment. There will be a 10% deduction for homework that is up to 24 hours late, and a 20% deduction for homework that is 24-48 hours late. We will not accept any homework that is more than 48 hours late.
Plan your time carefully, and don’t wait until the last minute to begin an assignment.
Determining the final grade:

- attendance/participation 10%
- assignments 40%
- midterm 17%
- final exam 33%

Extensions and makeup quizzes/exams will only be given in documented cases of serious illness or other emergencies.

You cannot redo or complete extra work to improve your grade.

Incompletes will not be given.

**Collaboration Policy***

The collaboration policy for this class is as follows.

- You are strongly encouraged to collaborate with one another in studying the textbook and the lecture material.

- As long as it satisfies the following conditions, collaboration on the homework assignments is encouraged and will not reduce your grade:

  1. Before discussing each homework problem with anyone else, you must give it an honest half-hour of serious thought.
  2. You may discuss ideas and approaches with other students in the class, but not share actual code or solutions to other types of problems. In other words, the work you submit must be entirely your own, which you must complete without looking at other people's work, and you must not permit others to copy your work. You must also acknowledge clearly in the appropriate portion of your solutions (e.g., in the comments of your code) people with whom you discussed ideas for that portion.
  3. You may get help from the teaching staff and from tutors in the lab for specific homework problems. Don't expect them to do it for you, however.
  4. If you get really stuck with a bug in a program (defined roughly as over an hour of frustration), you are allowed to get help from a friend as long as you acknowledge that help clearly in your solutions (e.g., in the comments of your code).
  5. You may not work with people outside this class (but come and talk to us if you have a tutor), seek online solutions, get someone else to do it for you, etc.

- You are not permitted to collaborate on exams.

The last point is particularly important: if you don't make an honest effort on the homework but always get ideas from others, your exam score will reflect it.

*Thanks to Prof. Leo Reyzin, who wrote the original versions of the sections describing the collaboration policy and violations to that policy. I have made only minor modifications.*
Violations of Collaboration Policy*
Violations of collaboration policy fall into two categories: ones that are *acknowledged* at the time they occur (for example, in clearly marked comments in your code) and ones that are *unacknowledged.*

Acknowledged violations (e.g., using someone else's code for a method you didn't know how to write yourself, and stating clearly in your code that this is not your own work) will result in an appropriate reduction in the grade, but will not be considered cheating. **You should send an email to cs460-staff@cs.bu.edu about all such violations, or anything that you think may possibly be considered as such.**

Unacknowledged violations of the collaboration policy—for example, not stating the names of your collaborators, or any other attempt to represent the work of another as your own—will result in an automatic failing grade and will be reported to the Academic Conduct Committee (ACC). The ACC often suspends or expels students deemed guilty of plagiarism or other forms of cheating. We will assume that you understand the CAS Academic Conduct Code, copies of which are available in CAS 105.

If you are uncertain as to whether a particular kind of interaction with someone else constitutes illegal collaboration or academic dishonesty, please ask Dr. Sullivan *before* taking any action that might violate the rules; if you can't reach him in time, then at the very least include a clear explanation of what happened in your homework to avoid being treated as a cheater. Citing your sources is usually the easiest way out of trouble.

**Textbooks**

- Computer Science 460/660 coursepack. This will be available for purchase at FedEx Office at the corner of Comm Ave. and Cummington. More information will be given during the first lecture.
- Optional readings will be also given from the following book: *Database Systems: The Complete Book (2nd edition)* by Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom (ISBN 978-0131873254, Pearson Prentice Hall, 2009). This book is *not* required, but you may find it useful to purchase it. Students in CS 660 will be required to do additional readings.
## Schedule (tentative)

<table>
<thead>
<tr>
<th>week</th>
<th>lecture dates</th>
<th>topics, exams, and special dates</th>
<th>optional readings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9/3, 9/5</td>
<td>introduction; database design and ER models</td>
<td>4.1-4</td>
</tr>
<tr>
<td>2</td>
<td>9/10, 9/12</td>
<td>the relational model relational algebra and SQL</td>
<td>2.1-5, 6.1-5</td>
</tr>
<tr>
<td>3</td>
<td>9/17, 9/19</td>
<td>more SQL</td>
<td>see above</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 1, part I due on 9/17</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9/16: last day to add a class</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9/24, 9/26</td>
<td>storage and indexing</td>
<td>13.5, 13.7,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 1, part II due on 9/24</strong></td>
<td>14.1-3</td>
</tr>
<tr>
<td>5</td>
<td>10/1, 10/3</td>
<td>the logical-to-physical mapping; key-value stores; query processing</td>
<td>15.1-3, 15.5-6</td>
</tr>
<tr>
<td>6</td>
<td>10/8, 10/10</td>
<td>transactions; concurrency control</td>
<td>6.6, 19.1-2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 2, part I due on 10/8</strong></td>
<td>18.1-8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10/7: last day to drop without a 'W', and to change from credit to audit status</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10/17</td>
<td>concurrency control (cont.)</td>
<td>see above</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 2, part II due on 10/15</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No lecture on Tuesday, 10/15 (Mon. schedule)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10/22, 10/24</td>
<td>recovery and logging</td>
<td>17.1-4</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Midterm exam on 10/24</strong></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10/29, 10/31</td>
<td>complex data; objects in database systems</td>
<td>4.9-10, 10.3-5</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 3, part I due on 10/29</strong></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11/5, 11/7</td>
<td>semistructured data; XML</td>
<td>11.1-4, 12.1-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 3, part II due on 11/5</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11/8: last day to drop a class with a 'W'</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11/12, 11/14</td>
<td>distributed databases and replication</td>
<td>20.3-4, 20.5-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 4, part I due on 11/12</strong></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11/19, 11/21</td>
<td>noSQL databases</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 4, part II due on 11/19</strong></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>11/26</td>
<td>graph databases</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No lecture on 11/28 (Thanksgiving recess)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>12/3, 12/5</td>
<td>performance tuning; other topics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 5, part I due 12/3</strong></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>12/10</td>
<td>wrap-up and review</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Problem Set 5, part II due on 12/10</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No lecture on 12/12 (Study period)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Final exam: Monday, 12/16, 9-11 a.m.</td>
<td></td>
</tr>
</tbody>
</table>
Welcome to CS 460/660

- This is a course on databases, but it's also more than that.

- We'll look at different ways of storing/managing data.

- Key lesson: there are multiple approaches to data-management problems.
  - one size doesn't fit all!

- Key goal: to be able to choose the right solution for a given problem.
Databases Are Everywhere

- Example collections of data:
  - account data: banks, credit-card companies, etc.
  - airline data: flights, reservations, etc.
  - student or employee data
  - biological data: DNA sequences, protein sequences, etc.
  - data collected by sensor networks
  - data in your personal address book

- Managed by some type of *database management system* (DBMS)

The Conventional Approach

- Use a DBMS that employs the relational model
  - RDBMS = *relational* database management system
  - view data as tables of records
  - 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

  \[
  \text{transaction } T1
  \begin{align*}
  \text{read } \text{balance}_1 \\
  \text{write}(\text{balance}_1 - 500) \\
  \text{write}(\text{balance}_2 + 500)
  \end{align*}
  \]

• Other examples:
  • making a flight reservation
    select flight, reserve seat, make payment
  • making an online purchase
  • making an ATM withdrawal

• 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 1:

    \[
    \text{transaction}
    \begin{align*}
    \text{read } \text{balance}_1 \\
    \text{write}(\text{balance}_1 - 500) \\
    \text{CRASH} \\
    \text{read } \text{balance}_2 \\
    \text{write}(\text{balance}_2 + 500)
    \end{align*}
    \]

    • what should happen?

  • example 2:

    \[
    \text{transaction}
    \begin{align*}
    \text{read } \text{balance}_1 \\
    \text{write}(\text{balance}_1 - 500) \\
    \text{read } \text{balance}_2 \\
    \text{write}(\text{balance}_2 + 500) \\
    \text{user told "transfer done"} \\
    \text{CRASH}
    \end{align*}
    \]

    • 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.

• May 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: use a key-value store
  - key = user id
  - value = password and other user information
- less overhead and easier to manage than an RDBMS
- still very powerful: transactions, recovery, replication, etc.

Example Problem II: Biological Data

- Long strings of data (e.g., DNA sequences)
  * gi|49175990|ref|NC_000913.2| Escherichia coli K12, complete genome
  * AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTA
  * AATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAA
  * ACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGA
  * CAGTGCGGGCTTTTTTTTTCGACCAAAGGTAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTC
  * TGCGTGTTGCCGATATTCTGGAAAGCAATGCCAGGCAGGGGCAGGTGGCCACCGTCCTCTCTGCCCCCGCCAAAATCACCAACCACCTGGTGGCGAT
  * GATTGAAAAAACCATTAGCGGCCAGGATGCTTTACCCAATATCAGCGATGCCGAACGTATTTTTGCCGAACTTTTGACGGGACTCGCCGCCGCCCAG
  * CCGGGGGTTTCCCGCTGGCGCAATTTGAACCTTTCTGCATCGAGAAATTGGCCCAATAAACACATCTCTCGATGCAATCAATTTTGCGGCGAGCGCC

- Exact matches are seldom found.
- Common queries involve looking for similarities or patterns.
  - what genes in mice are similar to genes in humans?
  - what genes contain the pattern TCTAGA?
- Solution: use text files and Python!
  - need special algorithms for finding statistically significant similarities
Example Problem III: Web Services

- Services provided or hosted by Google, Amazon, etc.
  - Google Analytics, Earth, Maps, Gmail, etc.
  - Netflix, Pinterest, Reddit, Flipboard, GitHub, etc.
- Can involve huge amounts of data / traffic
- Scalability is crucial
  - load can increase rapidly and unpredictably
  - use large clusters of commodity machines
- Conventional RDBMSs don't scale well in this way.
- Solution: some flavor of noSQL

One Size Doesn’t Fit All!

- noSQL = not only SQL
- Need to learn to choose the right tool for a given job.
- In some cases, may need to develop new tools!
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.

Options for the Logical Layer (partial list)

• relational model + SQL
• object-oriented model + associated query language
• XML + XPath or XQuery
• JSON + associated API
• key-value pairs + associated API
• graph-based model + associated API/query language
• comma-delimited or tab-delimited text + tool for text search
Options for the Storage Layer (partial list)

- transactional storage engine
  - supports transactions, recovery, etc.
- a non-transactional engine that stores data on disk
- an engine that stores data in memory
- a column store that stores columns separately from each other
  - vs. a traditional row-oriented approach
  - beneficial for things like analytical-processing workloads

Composing the Layers

- Example: MySQL
  - the RDBMS that we will use for the course
- MySQL's logical layer = relational model + SQL
- For the storage layer, MySQL provides multiple engines to choose from, including:
  - InnoDB: supports transactions
  - MyISAM: no transactions
Course Overview

- data models/representations (logical layer), including:
  - entity-relationship (ER): used in database design
  - relational (including SQL)
  - object-oriented and object-relational
  - XML
  - noSQL variants

- implementation issues (storage layer), including:
  - storage and indexing structures
  - transactions
  - concurrency control
  - logging and recovery
  - distributed databases and replication

Why Study DBMS Implementation?

- It’s interesting!

- Will allow you to make more intelligent choices about:
  - which approach to use for a particular problem
  - how to configure/tune a given system
Requirements

• Attendance and participation (10%)
  • every has an allowance of 2 missed classes; 
don’t email unless special circumstances

• Five homework assignments (40%)
  • each will have a "written" part and a "programming" part
  • CS 660 students will complete extra work
  • complete Problem Set 0 sometime this week
    (see course website)

• Midterm exam (17%)

• Final exam (33%)

Course Materials

• Lecture notes will be the primary resource.
  • coursepack with most of the notes will be at FedEx Office;
    the rest will be handed out in lecture

• Optional textbook: Database Systems: The Complete Book
  (2nd edition) by Garcia-Molina et al. (Prentice Hall, 2009)

• Other options:
  • Database Management Systems by Ramakrishnan and
    Gehrke (McGraw-Hill)
  • Database System Concepts by Silberschatz et al.
    (McGraw-Hill)

• CS 660 students will also read some database research papers.
Course Staff

- Instructor: Dave Sullivan
- TF: Haohan Zhu (zhu@cs.bu.edu)
- Office hours and contact info. will be available on the Web:
  http://www.cs.bu.edu/courses/cs460
- For questions on content, homework, etc.:
  • use Piazza
  • send e-mail to cs460-staff@cs.bu.edu

Other Details of the Syllabus

- Collaboration
- Policies:
  • lateness
  • please don't request an extension unless it's an emergency!
  • grading
- Please read the syllabus carefully and make sure that you understand the policies and follow them carefully.
- Let us know if you have any questions.
Algorithm for Finding My Office

1. Go to the entrance to the MCS (math/CS) building at 111 Cummington Street – behind Warren Towers.  
   Do not enter this building!

2. Turn around and cross the street to the doors across from MCS.

3. Enter those doors and take an immediate right.  
   (continued on next slide)

Algorithm for Finding My Office (cont.)

4. As you turn right, you should see the door below. Open it and go up to the second floor.

5. As you leave the stairs, turn right and then go left into a small hallway. My office is the first door on the left (PSY 228D).
Database Design
and the Entity-Relationship Model

Computer Science 460/660
Boston University
Fall 2013
David G. Sullivan, Ph.D.

Database Design

• In database design, we determine:
  • which data fields 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

- It can sometimes make sense to combine seemingly distinct types of entities.

  - example:

    Student \(\rightarrow\) Person
    Employee \(\rightarrow\)

    because an employee can also be a student

  - why might it beneficial to combine them?

Entities: the “Things” (cont.)

- 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
  - dotted oval = attribute that can be derived from other attributes

```
Course
    name
    room
    start time
    end time
    exam dates
    length = end time - start time
```

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:

```
Person
    id
    name
    dob
    email
```

- possible keys include:
  - 
  - 
  - 
  - 

CAS CS 460/660, Fall 2013

David G. Sullivan, Ph.D.
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: consider again our Person entity set
  - (name, dob) is a minimal key if we need both attributes to uniquely identify a person
  - (id, email) is a key, but it is not minimal, because just one of these attributes is sufficient

Candidate Key (cont.)

- Consider an entity set for books:

  (assume that an author does not write two books with the same title)

  **Book**
  - isbn
  - author
  - title

  key?  candidate key?

  author

  author, title

  isbn, author
**Primary Key**

- When defining a relation, we typically choose one of the candidate keys as the *primary key*.
- The records are arranged on disk to allow for quick retrieval using the value of the primary key.
- In an ER diagram, we underline the primary key attribute(s).

![Course diagram with attributes: name, room, start time, end time, exam dates, length]

**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
  
  ![Person Enrolled Course diagram]

- Another example: courses meet in rooms
  
  ![Course Meets In Room diagram]
Relationships Between Entities (cont.)

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

  ![Diagram of relationships between Course, Meets In, and Room]

- 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 diagram: Person Enrolled Course with credit status]

- Can you think of an attribute for Meets In?

  ![Diagram of Course Meets In Room with a question mark]
What courses is student 45678900 enrolled in, and for what credit status?
Note: For a given combination of entities, there can be at most one value for each relationship-set attribute.

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

  ![Diagram of Degree 2 Relationship Set]

- It's also possible to have higher-degree relationship sets.

- A ternary relationship set connects three relationship sets.
  - degree = 3

  ![Diagram of Degree 3 Relationship Set]

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.

  ![Diagram of Role Indicators]

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

- Relationships like this one are referred to as recursive relationships.
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 room

- A key constraint specifies a functional mapping from one entity set to another.
  - each course is mapped to at most one room (course \(\rightarrow\) 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 from Course to Room
- 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.

Other Examples

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

- Majors In is what type of relationship in this case?
Other Examples (cont.)

• What if each student can have more than one major?

- Person \(\text{Majors In}\) Department

• Majors In is what type of relationship in this case?

Other Examples (cont.)

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

- Person \(\text{advisor}\) \(\text{Advises}\) advisee

• Advises is what type of relationship?
Review: Cardinality Constraints

- many-to-one relationship
  ![Diagram of many-to-one relationship]

- one-to-one relationship
  ![Diagram of one-to-one relationship]

- many-to-many relationship
  ![Diagram of many-to-many relationship]

Cardinality Constraints and Ternary Relationship Sets

- The arrow into “study group” encodes the following constraint:
  “a student studies in at most one study group for a given course.”

- In other words, a given (student, course) combination is mapped to at most one study group.
  - thus, a given (student, course) combination can appear in at most one row of the corresponding table
  - a given student or course can itself appear in multiple rows
Other Details of Cardinality Constraints

- For ternary and higher-degree rel. sets, we limit ourselves to a single arrow, since otherwise the meaning can be ambiguous.

```
study group

person ─ studies in ─ course
```

- For example, the diagram above could mean:
  1) person is mapped to at most one (course, study group) combo
  2) each (person, course) combo is mapped to at most one study group

  and

  each (person, study group) combo is mapped to at most one course

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.
  - indicate using a thick line (or double line)

- Example: each department must have at least one chairperson.

```
Person ─ Chairs ─ Department
```

- 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 *exactly one* person as a chair
  - arrow into Person specifies at most one
  - thick line from Dept to Chairs specifies at least one
  - at most one + at least one = exactly one

Design Issue: Attribute or Entity Set?

- It can sometimes be hard to decide if something should be treated as an attribute or an entity set. Example:

  - Indications that you should use an entity set:
    - if it has attributes of its own that you wish to capture
    - if, as an attribute, it could have multiple values
      - you could potentially use a multi-valued attribute, but you end up treating it as an entity set when converting to a relational model
Design Issue: Relationship Set or Entity Set?

- Example: students belong to study groups for courses

- Which design is better? Why?

- Rule of thumb: use relationship sets for actions between entities.

Design Issue: Relationship Set or Entity Set? (cont.)

- What if a student can belong to more than one study group for a given course?
What Is a Data Model?

- A formal way of describing a database:
  - the pieces of data (*data items*)
  - relationships between data items
  - constraints on the values of data items

- We'll focus today on the relational model – the dominant data model in current database systems.

- To understand the benefits of the relational model, it helps to briefly consider earlier models.
Earlier Data Models

• Before the relational model, the data models were closely tied to the physical representation of the data.

• To access data records, users had to write programs that navigated from one record to another.
  • difficult to write
  • adding new fields required modifying the programs – even if the programs were not accessing the new fields!

The Relational Model: A Brief History


• The 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: A Brief History (cont.)

"Codd had a bunch of ... fairly complicated queries....
I could imagine how those queries would have been represented ... by programs that were five pages long....
Codd would sort of write them down as one-liners ... they weren't complicated at all.
I said, 'Wow.' This was kind of a conversion experience for me."

— Don Chamberlin, describing a seminar that Codd gave at IBM about the new model

• Codd won the Turing Award (computer science's Nobel Prize) in 1981 for his work.

The Relational Model: Basic Concepts

• A database consists of a collection of tables.

• Example of a table:

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

• 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
Mathematical Foundations: Cartesian Product

• Let: $A$ be the set of values $\{a_1, a_2, \ldots\}$
  $B$ be the set of values $\{b_1, b_2, \ldots\}$

• The \textit{Cartesian product} of $A$ and $B$ (written $A \times B$) is the set of all possible ordered pairs $(a_i, b_j)$, where $a_i \in A$ and $b_j \in B$.

• Example:
  $A = \{\text{apple, pear, orange}\}$
  $B = \{\text{cat, dog}\}$
  $A \times B = \{(\text{apple, cat)}, (\text{apple, dog}), (\text{pear, cat}), (\text{pear, dog}), (\text{orange, cat}), (\text{orange, dog})\}$

• Example:
  $C = \{5, 10\}$
  $D = \{2, 4\}$
  $C \times D = ?$

Mathematical Foundations: Cartesian Product (cont.)

• We can also take the Cartesian product of three of more sets.

• $A \times B \times C$ 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:
    $C = \{5, 10\}$
    $D = \{2, 4\}$
    $E = \{\text{"hi", "there"}\}$
    $C \times D \times E = \{(5, 2, "hi"), (5, 2, "there"), (5, 4, "hi"), (5, 4, "there"), (10, 2, "hi"), (10, 2, "there"), (10, 4, "hi"), (10, 4, "there")\}$

• $A_1 \times A_2 \times \ldots \times A_n$ is the set of all possible ordered \textit{tuples} $(a_{1i}, a_{2j}, \ldots, a_{nk})$, where $a_{di} \in A_d$. 
Mathematical Foundations: Relations

• A relation on a set of variables is a subset of the Cartesian product of the domains of the variables.

• Example: let x and y be variables that both have the set of non-negative integers as their domain.
  • \{(2, 5), (3, 10), (13, 2), (6, 10)\} is one relation on (x, y)
  • \{(0, 1), (5, 3)\} is another

• In the relational model, a table is a subset of the Cartesian product of the domains of its columns.
  { (12345678, "Jill Jones", "Canaday C-54", 2011, 3/10/85),
    (25252525, "Alan Turing", "Lowell F-51", 2008, 2/7/88),
    (33566891, "Audrey Chu", "Pfoho, Moors 212", 2009, 10/2/86), ...
  }  

• Thus, a table is a relation!

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 one of the following:
  - 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:
  \[ \text{Student}(id, \text{name}, \text{address}, \text{email}, \text{phone}) \]

• 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 \( \rightarrow \) a relation with the same attributes
  • relationship set \( \rightarrow \) a relation whose attributes are:
    • the primary keys of the connected entity sets
    • the descriptive attributes of the relationship set

• Example of converting a relationship set:

\[ \text{Enrolled}(id, \text{name}, \text{credit\_status}) \]

• 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:

  ![Diagram](image)

  \[
  \text{MeetsIn}(\text{name}, \text{name})
  \]

  \[
  \text{MeetsIn}(\text{course}, \text{room})
  \]

  - 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:

  ![Diagram](image)

  \[
  \text{Course}(\text{name, start\_time, end\_time})
  \]

  \[
  \text{Room}(\text{name, capacity})
  \]

  \[
  \text{MeetsIn}(\text{course, room})
  \]
Special Case: Many-to-One Relationship Sets (cont.)

- However, if a relationship set is many-to-one, we often:
  - use only two relations – one for each of the entity sets
  - capture the relationship set in the relation used for the entity set on the *many* side of the relationship

\[\text{Course}(\text{name}, \text{start\_time, end\_time, room})\]
\[\text{Room}(\text{name, capacity})\]

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

- Advantages of this approach?
  - 
  -

<table>
<thead>
<tr>
<th>Course</th>
<th>MeetsIn</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>course-name</td>
<td>course-name</td>
<td>room-name</td>
</tr>
<tr>
<td>csci50b</td>
<td>csci50b</td>
<td>Sci Ctr B</td>
</tr>
<tr>
<td>csci119</td>
<td>csci119</td>
<td>Sever 213</td>
</tr>
<tr>
<td>csci160</td>
<td>csci160</td>
<td>Sci Ctr A</td>
</tr>
<tr>
<td>csci268</td>
<td>csci268</td>
<td>Sci Ctr A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>course-name</td>
<td>room-name</td>
</tr>
<tr>
<td>csci50b</td>
<td>Sci Ctr B</td>
</tr>
<tr>
<td>csci119</td>
<td>Sever 213</td>
</tr>
<tr>
<td>csci160</td>
<td>Sci Ctr A</td>
</tr>
<tr>
<td>csci268</td>
<td>Sci Ctr A</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>course-name</th>
<th>room-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscie50b</td>
<td>Sci Ctr B</td>
</tr>
<tr>
<td>cscie119</td>
<td>Saver 213</td>
</tr>
<tr>
<td>cscie160</td>
<td>Sci Ctr A</td>
</tr>
<tr>
<td>cscie268</td>
<td>Sci Ctr A</td>
</tr>
<tr>
<td>cscie160</td>
<td>NULL</td>
</tr>
</tbody>
</table>

- 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:

  ![Diagram](image.png)

  - Which of these would probably make more sense?
Many-to-Many Relationship Sets

- For many-to-many relationship sets, it's better to stick with a separate relation for the relationship set.
- Otherwise, we can end up with redundant information.
  - example:

  ![Diagram](image)

  - trying to capture Enrolled in Student gives:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>...</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>112345678</td>
<td>jill jones</td>
<td>51 Brattle St.</td>
<td>...</td>
<td>CS 105</td>
</tr>
<tr>
<td>112345678</td>
<td>jill jones</td>
<td>51 Brattle St.</td>
<td>...</td>
<td>Math 120</td>
</tr>
<tr>
<td>112345678</td>
<td>jill jones</td>
<td>51 Brattle St.</td>
<td>...</td>
<td>English 145</td>
</tr>
</tbody>
</table>

Primary Keys of Relations for Entity Sets

- When translating an entity set to a relation, the primary key of the entity set becomes the primary key of the relation.

  ![Diagram](image)
Primary Keys of Relations for Relationship Sets

- When translating a relationship set to a relation, the primary key of the resulting relation depends on the cardinality constraints specified in the ER diagram.

- If the relationship set is many-to-many, we form the union of the primary keys of the connected entity sets.

  \[ \text{Enrolled}(id, name, credit\_status) \]
  \[ \text{Enrolled(student\_id, course\_name, credit\_status)} \]

  • why is this necessary?

Primary Keys of Relations for Relationship Sets (cont.)

- If the relationship set is many-to-one, the primary key includes only the primary key of the entity set at the many end.

  \[ \text{Borrows(id, isbn)} \]
  \[ \text{Borrows(person, isbn)} \]

  • we need to limit the primary key to isbn in order to capture the cardinality constraint. why?

  • how else could we capture this relationship set?
Primary Keys of Relations for Relationship Sets (cont.)

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

  ![Diagram](https://via.placeholder.com/150)

  \[ \text{Holds}(id, id) \]

  \[ \text{Holds}(\text{person}, \text{chair}) \]

Foreign Keys

- A **foreign key** is attribute(s) in one relation that take on values from the primary-key attribute(s) of another (foreign) relation.

- Example: **MajorsIn** has **two** foreign keys.

  ![Table](https://via.placeholder.com/150)

  - **Student**
    - id
    - name
    - computer science
    - english
    - ...

  - **Department**
    - name
    - computer science
    - english
    - ...

- We use foreign keys to capture relationships between entities.
Constraints

• Specifying a primary key imposes a uniqueness constraint: no combination of values of the PK attributes can appear more than once.

• Specifying a foreign key imposes a referential integrity constraint: the FK attribute(s) must take on values that appear in the corresponding PK attribute(s).

Constraints (cont.)

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


<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>computer science</td>
</tr>
<tr>
<td>12345678</td>
<td></td>
<td>english</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
</tbody>
</table>

• 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

• The query language proposed by Codd.
  • a collection of operations on relations

• For each operation, both the operands and the result are relations.

one or more relations → operation → a relation

• Relational algebra treats relations as sets.
  If an operation creates duplicate tuples, they are removed.

• We’ll cover just enough to help us understand SQL queries
  and to write simple relational algebra queries.

Selection

• What it does: selects tuples from a relation that match a predicate
  • predicate = condition

• Syntax: \( \sigma_{\text{predicate}}(\text{relation}) \)

• Example: Enrolled

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>csci e50b</td>
<td>undergrad</td>
</tr>
<tr>
<td>25252525</td>
<td>csci e160</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>csci e268</td>
<td>graduate</td>
</tr>
<tr>
<td>33566891</td>
<td>csci e119</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>csci e119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

\( \sigma_{\text{credit_status} = \text{"graduate"}}(\text{Enrolled}) = \)

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>45678900</td>
<td>csci e268</td>
<td>graduate</td>
</tr>
<tr>
<td>45678900</td>
<td>csci e119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie160</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{student_id}, \text{credit_status}}(\text{Enrolled}) = \]

- duplicates, so we keep only one

Combining Operations

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

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie160</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{student_id}, \text{credit_status}}(\sigma_{\text{credit_status} = \text{graduate}}(\text{Enrolled})) = \]

- duplicates, so we keep only one
Set Operations

- **Union**: $A \cup B = \text{all tuples in } A \text{ and/or } B$
- **Intersection**: $A \cap B = \text{all tuples in both } A \text{ and } B$
- **Set Difference**: $A - B = \text{tuples in } A \text{ but not in } B$

Example: Enrolled *(note: slightly different from the one in the earlier slides)*

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

$\text{Enrolled} - \sigma_{\text{credit_status}=\text{"undergrad"}}(\text{Enrolled})$

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

Set Operations (cont.)

- **Example of where set difference is used:**
  Of the students enrolled in courses, which ones are not enrolled in any courses for graduate credit?

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

- The following query does *not* work. Why?
  $$\pi_{\text{student_id}}(\sigma_{\text{credit_status} = \text{"graduate"}}(\text{Enrolled}))$$

- This query *does* work:
  $$\pi_{\text{student_id}}(\text{Enrolled}) - \pi_{\text{student_id}}(\sigma_{\text{credit_status} = \text{"graduate"}}(\text{Enrolled}))$$
**Cartesian Product**

- **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 \( R_2 \)
  - if there are two attributes with the same name, we prepend the name of the original relation
  - example: the attributes of \( \text{Enrolled} \times \text{MajorsIn} \) would be
    
    \[
    (\text{Enrolled}.\text{student_id}, \text{course_name}, \text{credit_status}, \text{MajorsIn}.\text{student_id}, \text{department})
    \]

**Cartesian Product (cont.)**

- **Example:**

<table>
<thead>
<tr>
<th>Enrolled</th>
<th>MajorsIn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>student_id</strong></td>
<td><strong>department</strong></td>
</tr>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>comp sci</td>
</tr>
<tr>
<td>33566891</td>
<td>english</td>
</tr>
<tr>
<td>25252525</td>
<td>the occult</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enrolled x MajorsIn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>student_id</strong></td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>45678900</td>
</tr>
<tr>
<td>45678900</td>
</tr>
</tbody>
</table>
**Rename**

- **What it does:** gives a (possibly new) name to a relation, and optionally to its attributes
  - common use: naming the relation that is the result of one or more operations
- **Syntax:**
  \[
  \rho_{\text{rel} \_\text{name}}(\text{relation}) \\
  \rho_{\text{rel} \_\text{name}(A_1, A_2, \ldots, A_n)}(\text{relation})
  \]
- **Examples:**
  - renaming to allow us to take the Cartesian product of a relation with itself:
    \[
    \rho_{E_1}(\text{Enrolled}) \times \rho_{E_2}(\text{Enrolled})
    \]
  - renaming to give a name to the result of an operation:
    \[
    \sigma_{\text{room} = \text{BigRoom} \_\text{name}}(\text{Course} \times \rho_{\text{BigRoom}}(\sigma_{\text{capacity} > 200}(\text{Room})))
    \]

  What does this query give us? Assume this schema:
  - Course(name, start\_time, end\_time, room)
  - Room(name, capacity)

**Natural Join**

- **What it does:** performs a “filtered” Cartesian product
  - filters out tuples in which attributes with the same name have different values
- **Syntax:** \( R_1 \bowtie R_2 \)
- **Example:**

  \[
  \begin{array}{ccc}
  g & h & i \\
  \text{foo} & 10 & 4 \\
  \text{bar} & 20 & 5 \\
  \text{baz} & 30 & 6 \\
  \end{array}
  \quad \begin{array}{ccc}
  i & j & g \\
  4 & 100 & \text{foo} \\
  4 & 300 & \text{bop} \\
  5 & 400 & \text{baz} \\
  5 & 600 & \text{bar} \\
  \end{array}
  \]

  \[
  \begin{array}{ccc}
  g & h & i \\
  \text{foo} & 10 & 4 \\
  \text{bar} & 20 & 5 \\
  \end{array}
  \quad \begin{array}{ccc}
  i & j & g \\
  100 & \text{foo} \\
  600 & \text{bar} \\
  \end{array}
  \]

  \[
  \begin{array}{ccc}
  g & h & i \\
  \text{foo} & 10 & 4 \\
  \text{bar} & 20 & 5 \\
  \end{array}
  \quad \begin{array}{ccc}
  i & j & g \\
  100 & \text{foo} \\
  600 & \text{bar} \\
  \end{array}
  \]
Performing the Natural Join

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

- **Example**:

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_1 \times R_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
</tr>
<tr>
<td>foo 10</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>bar 20</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>baz 30</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Performing the Natural Join

- **Step 2**: perform a selection in which we filter 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**:

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_1 \times R_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g$</td>
<td>$h$</td>
<td>$i$</td>
</tr>
<tr>
<td>foo 10</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>bar 20</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>baz 30</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
Performing the Natural Join

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

- **Example**:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>bar</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>baz</td>
<td>30</td>
<td>6</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>bar</td>
<td>4</td>
<td>300</td>
</tr>
<tr>
<td>baz</td>
<td>5</td>
<td>400</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>bar</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>baz</td>
<td>5</td>
<td>600</td>
</tr>
</tbody>
</table>

Performing the Natural Join

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

- **Example**:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>bar</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>baz</td>
<td>30</td>
<td>6</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>bar</td>
<td>4</td>
<td>600</td>
</tr>
<tr>
<td>baz</td>
<td>5</td>
<td>600</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>bar</td>
<td>5</td>
<td>600</td>
</tr>
</tbody>
</table>
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 \( R_1 \times R_2 \), but with common attributes included only once
  \[(a, b, c) \times (a, d, c, f) \rightarrow (a, b, c, d, f)\]

- If there are no common attributes, \( R_1 \bowtie R_2 = R_1 \times R_2 \)

Condition Joins (“Theta Joins”)

- **What it does:** performs a "filtered" Cartesian product according to a specified predicate
- **Syntax:** \( R_1 \bowtie_\theta R_2 \), where \( \theta \) is a predicate
- **Fundamental-operation equivalent:** cross, select using \( \theta \)
- **Example:**

<table>
<thead>
<tr>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_1 \bowtie(d &gt; c) R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( b )</td>
<td>( c )</td>
</tr>
<tr>
<td>foo</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>bar</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>baz</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>foo</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>bar</td>
<td>20</td>
<td>5</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
<th>student_id</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
<td>66666666</td>
<td>the occult</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
<td>33566891</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
<td>25252525</td>
<td></td>
</tr>
</tbody>
</table>

Enrolled \(\bowtie\) MajorsIn

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
<td>null</td>
</tr>
</tbody>
</table>

• Why isn’t this sufficient?

Outer Joins

• Outer joins allow us to include unmatched tuples in the result.
• Left outer join \(R_1 \bowtie R_2\): 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 \(\bowtie\) MajorsIn

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
<td>null</td>
</tr>
</tbody>
</table>

Enrolled \(\bowtie\) MajorsIn

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
<td>null</td>
</tr>
</tbody>
</table>
### Outer Joins (cont.)

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

#### Enrolled MajorsIn

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
<td>comp sci</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
<td></td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_id</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>comp sci</td>
</tr>
<tr>
<td>90765432</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

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

#### Enrolled MajorsIn

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
<td></td>
</tr>
<tr>
<td>90765432</td>
<td>null</td>
<td>null</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>null</td>
<td>null</td>
<td>the occult</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_id</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>comp sci</td>
</tr>
<tr>
<td>90765432</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>
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:**
  - simplifying complex expressions
    - example: recall this expression
      $$\sigma_{\text{room}} = \text{BigRoom.name} \ (\text{Course} \times \rho_{\text{BigRoom}} (\sigma_{\text{capacity} > 200} (\text{Room})))$$
    - simpler version using assignment:
      $$\text{BigRoom} \leftarrow \sigma_{\text{capacity} > 200} (\text{Room})$$
      $$\sigma_{\text{room}} = \text{BigRoom.name} \ (\text{Course} \times \text{BigRoom})$$

  - modifications (although we won’t cover this)

Union

- **What it does:** combines two relations

- **Syntax:** $R_1 \cup R_2$

- **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
• **Two examples:**

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>csci50b</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>csci160</td>
<td>undergrad</td>
</tr>
<tr>
<td>45678900</td>
<td>csci268</td>
<td>graduate</td>
</tr>
<tr>
<td>33566891</td>
<td>csci119</td>
<td>non-credit</td>
</tr>
<tr>
<td>25252525</td>
<td>csci119</td>
<td>graduate</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{student_id}}(\text{Enrolled}) \cup \pi_{\text{student_id}}(\text{MajorsIn}) \]

\[
\begin{array}{c|c|c}
\text{student_id} & \text{course_name} & \text{credit_status} \\
\hline
12345678 & csci50b & undergrad \\
45678900 & csci160 & undergrad \\
45678900 & csci268 & graduate \\
33566891 & csci119 & non-credit \\
25252525 & csci119 & graduate \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{student_id} & \text{department} \\
\hline
12345678 & comp sci \\
45678900 & mathematics \\
33566891 & comp sci \\
98765432 & english \\
66666666 & the occult \\
\end{array}
\]

\[
\pi_{\text{student_id}}(\text{MajorsIn}) - \pi_{\text{student_id}}(\text{Enrolled})
\]

\[
\begin{array}{c|c|c}
\text{student_id} & \text{department} \\
\hline
98765432 & english \\
66666666 & the occult \\
\end{array}
\]
Intersection

- **What it does:** selects tuples that are in both of two relations
- **Syntax:** \( R_1 \cap R_2 \)
- **Rules:** same as for union
- **Fundamental-operation equivalent:** \( R_1 \cap R_2 = R_1 - (R_1 - R_2) \)
- **Example:**

<table>
<thead>
<tr>
<th>student_id</th>
<th>course_name</th>
<th>credit_status</th>
<th>student_id</th>
<th>department</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie50b</td>
<td>undergrad</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie160</td>
<td>undergrad</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie268</td>
<td>graduate</td>
<td>33566891</td>
<td>comp sci</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie119</td>
<td>non-credit</td>
<td>98765432</td>
<td>english</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>graduate</td>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

\( \pi_{\text{student_id}}(\text{Enrolled}) \cap \pi_{\text{student_id}}(\sigma_{\text{department} = \text{"comp sci"}}(\text{MajorsIn})) \)

<table>
<thead>
<tr>
<th>student_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>33566891</td>
</tr>
</tbody>
</table>
Example Domain: a University

- We’ll continue to use relations from a university database.
  - 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)
  - two relations that capture relationships between entities:
    - MajorsIn(student, dept)
    - Enrolled(student, course, credit_status)
- The Course relation also captures a relationship – the relationship between a course and the room in which it meets.
- A few slides use additional relations.
- Note the difference: Course vs. course, Student vs. student
Querying an Existing Database

- In the problem set, we give you the commands needed to create the database that you will use.

- Thus, we'll focus first on how to perform queries on an existing database.

- Later, we'll look at how to:
  - create a table (building on what we covered earlier)
  - modify or remove an existing table
SELECT

• Used to implement most of the relational-algebra operations

• **Basic syntax:**
  
  ```
  SELECT a_1, a_2, ...
  FROM R_1, R_2, ...
  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 \( a_1, a_2, \ldots \) 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, course, credit_status)`
  
  `MajorsIn(student, dept)`

  we want find the major of the student Alan Turing.

• Here’s a query that will give us the answer:

  ```
  SELECT dept
  FROM Student, MajorsIn
  WHERE name = 'Alan Turing'
  AND id = student;
  ```

• The next two slides show how steps 1-3 from the previous slide combine to produce the desired result.
### Student

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

### MajorsIn

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>66666666</td>
<td>the occult</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>45678900</td>
<td>english</td>
</tr>
</tbody>
</table>

### SELECT dept
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student;

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>25252525</td>
<td>comp sci</td>
</tr>
</tbody>
</table>

### SELECT dept
FROM Student, MajorsIn
WHERE name = 'Alan Turing' AND id = student;

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>25252525</td>
<td>comp sci</td>
</tr>
</tbody>
</table>
Join Conditions

- Here’s the query from the previous problem:

```
SELECT dept
FROM Student, MajorsIn
WHERE name = 'Alan Turing'
    AND id = student;
```

- `id = student` 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.

<table>
<thead>
<tr>
<th>Student</th>
<th>name</th>
<th>MajorsIn</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
<td>56789000</td>
<td>mathematics</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
<td>56789000</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
<td>56789000</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
<td>66666666</td>
<td>the occult</td>
</tr>
</tbody>
</table>

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
FROM Enrolled;
```

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie268</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>cs165</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie119</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie268</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie275</td>
<td>grad</td>
</tr>
<tr>
<td>student</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345678</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25252525</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45678900</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33566891</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45678900</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

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

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>csci e268</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>cs165</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>csci e119</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>csci e268</td>
<td>non-credit</td>
</tr>
<tr>
<td>45678900</td>
<td>csci e275</td>
<td>grad</td>
</tr>
</tbody>
</table>

Another Example

- Given these relations:
  
  Student(id, name)
  Enrolled(student, course, credit_status)
  MajorsIn(student, dept)

- Find the names of all students enrolled in CSCI E-268 with undergrad status:
The WHERE Clause

```sql
SELECT column1, column2, ...
FROM table
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

String Comparisons

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

- In MySQL, comparisons involving strings are not case sensitive by default.
- to get case-sensitive comparisons, use the keyword BINARY before at least one of the two values
  ```
  'abc' = 'ABC' is true
  'abc = BINARY 'ABC' is false
  ```
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

• We can 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

• Here's how to find the names and rooms of CS courses:

  SELECT name, room
  FROM course
  WHERE name LIKE 'cs%';

Comparisons Involving Pattern Matching (cont.)

• More examples: let's say that you have the comparison
  word LIKE '_at%'

• what is the value of this comparison (true or false) for each of the following values of the column word?

<table>
<thead>
<tr>
<th>'batter'</th>
<th>'attribute'</th>
<th>'at'</th>
<th>'atomic matter'</th>
<th>'sat'</th>
</tr>
</thead>
</table>
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
  room != NULL
  NULL != 10
  NULL = NULL
  ```

- To test for the presence or absence of a NULL value, SQL provides special operators called IS NULL and IS NOT NULL.

- Example:
  ```
  SELECT name
  FROM Course
  WHERE room IS NULL;
  ```

---

The SELECT Clause

```sql
SELECT column1, column2, ... 
FROM table
WHERE selection condition;
```

- In addition to column names, can include constants/expressions:
  ```
  SELECT 'final exam', name, points/300*100
  ```

- Removing duplicates:
  - by default, the relation produced by a SELECT command may include duplicate tuples
  - to eliminate duplicates, add the DISTINCT keyword:
    ```
    SELECT DISTINCT column1, column2, ...
    ```
More Example Queries

- Given these relations:
  - Student(id, name)
  - Enrolled(student, course, credit_status)
  - MajorsIn(student, dept)

- Find the names of all students enrolled in any course:

- Find the name and credit status of all students enrolled in cs165 who are majoring in computer science:

Renaming Attributes or Tables

- Use the keyword AS

- Example:
  
  ```sql
  SELECT name AS student, credit_status AS student
  FROM Student, Enrolled AS E, MajorsIn AS M
  WHERE id = E.student
  AND E.student = M.student
  AND course = 'cs165'
  AND dept = 'comp sci';
  ```

<table>
<thead>
<tr>
<th>student</th>
<th>credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jill Jones</td>
<td>undergrad</td>
</tr>
<tr>
<td>Alan Turing</td>
<td>non-credit</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Renaming allows us to cross a relation with itself:

  ```sql
  SELECT E1.name AS student
  FROM Employee AS E1, Employee AS E2
  WHERE E1.supervisor = E2.id
  AND E2.name = 'Teresa Lopes';
  ```
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:

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

<table>
<thead>
<tr>
<th>Room</th>
<th>id</th>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Sanders Theatre</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>Sever 111</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>Sever 213</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>Sci Ctr A</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>Sci Ctr B</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td>Emerson 105</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td>Sci Ctr 110</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

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:

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

- **COUNT( attribute)** counts the number of non-NULL values of attribute, so it won't always be equivalent to COUNT(*).

- Aggregate functions **cannot** be used in the WHERE clause.

- Practice with aggregate functions: write a query to find the largest capacity of any room in the Science Center:

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

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

---

### Room Table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Sanders Theatre</td>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
<td>Sever 111</td>
<td>50</td>
</tr>
<tr>
<td>3000</td>
<td>Sever 213</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>Sci Ctr A</td>
<td>300</td>
</tr>
<tr>
<td>5000</td>
<td>Sci Ctr B</td>
<td>500</td>
</tr>
<tr>
<td>6000</td>
<td>Emerson 105</td>
<td>500</td>
</tr>
<tr>
<td>7000</td>
<td>Sci Ctr 110</td>
<td>30</td>
</tr>
</tbody>
</table>

---

### Room Table with MAX Capacity

<table>
<thead>
<tr>
<th>name</th>
<th>MAX(capacity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sci Ctr A</td>
<td>300</td>
</tr>
<tr>
<td>Sci Ctr B</td>
<td>500</td>
</tr>
<tr>
<td>Sci Ctr 110</td>
<td>30</td>
</tr>
</tbody>
</table>
Subqueries

• A subquery allows us to use the result of one query in the evaluation of another query.
  • the queries can involve the same table or different tables

• We can use a subquery to solve the previous problem:

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

the subquery

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

Subqueries and Set Comparisons

• Subqueries can be used to perform comparisons with elements of a set using the ALL and SOME operators.
  • example: find all rooms whose capacity is bigger than that of all rooms in Sever Hall.

```sql
SELECT name, capacity
FROM Room
WHERE capacity > ALL (SELECT capacity
                      FROM Room
                      WHERE name LIKE 'Sever%');
```

• example: find all rooms whose capacity is greater than that of some room in Sever Hall.

```sql
SELECT name, capacity
FROM Room
WHERE capacity > SOME (SELECT capacity
                        FROM Room
                        WHERE name LIKE 'Sever%');
```
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 enrolled in CSCI E-268
    ```sql
    SELECT name
    FROM Student
    WHERE id IN (SELECT student
        FROM Enrolled
        WHERE course = 'csci268');
    ```

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:
  ```sql
  SELECT course, COUNT(*)
  FROM Enrolled
  GROUP BY course;
  ```

- When you group by an attribute, you can 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, COUNT(*)
FROM Enrolled
GROUP BY course;
```

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie268</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie119</td>
<td>grad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie275</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie268</td>
<td>non-credit</td>
</tr>
<tr>
<td>66666666</td>
<td>cscie268</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>ugrad</td>
</tr>
</tbody>
</table>

![Diagram of query evaluation]

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

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie268</td>
<td>ugrad</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie268</td>
<td>non-credit</td>
</tr>
<tr>
<td>66666666</td>
<td>cscie268</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie119</td>
<td>grad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie275</td>
<td>grad</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>ugrad</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscie268</td>
<td>3</td>
</tr>
<tr>
<td>cscie119</td>
<td>2</td>
</tr>
<tr>
<td>cscie275</td>
<td>1</td>
</tr>
</tbody>
</table>

Enrolled result of the query

- Important difference:
  - A `WHERE` clause is applied before grouping
  - A `HAVING` clause is applied after grouping
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
    ```sql
    SELECT AVG(count)
    FROM (SELECT course, COUNT(*) as count
          FROM Enrolled
          GROUP BY course) AS enrollCounts
    WHERE course LIKE 'cs%';
    ```

- Some systems require that you assign a FROM-clause subquery a name (e.g., enrollCounts above).

---

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, use DESC to get descending
  - example:
    ```sql
    SELECT name, capacity
    FROM Room
    WHERE capacity > 100
    ORDER BY capacity DESC, name;
    ```

<table>
<thead>
<tr>
<th>name</th>
<th>capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sanders Theatre</td>
<td>1000</td>
</tr>
<tr>
<td>Emerson 105</td>
<td>500</td>
</tr>
<tr>
<td>Sci Ctr B</td>
<td>500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Set Operations

- **UNION**
- **INTERSECTION**
- **EXCEPT** (set difference)

- Example: find the IDs of students and advisors
  
  ```sql
  SELECT student
  FROM Enrolled
  UNION
  SELECT advisor
  FROM Advises;
  ```

Outer Joins

- Syntax for left outer join:
  ```sql
  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 tuples in the Cartesian product that satisfy the join condition in the **ON** clause
  - including an extra tuple for each row from T1 that does not have a match with a row from T2
  - the T2 attributes in the extra tuples are given null values
  - applying the remaining clauses as before

- Also available: **RIGHT OUTER JOIN**, **FULL OUTER JOIN**
Outer Joins (cont.)

- Example: get the IDs and majors of all enrolled students.

```
<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>credit_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>cscie268</td>
<td>ugrad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie119</td>
<td>grad</td>
</tr>
<tr>
<td>45678900</td>
<td>cscie275</td>
<td>grad</td>
</tr>
<tr>
<td>33566891</td>
<td>cscie268</td>
<td>non-credit</td>
</tr>
<tr>
<td>66666666</td>
<td>cscie268</td>
<td>ugrad</td>
</tr>
<tr>
<td>25252525</td>
<td>cscie119</td>
<td>ugrad</td>
</tr>
</tbody>
</table>
```

```
SELECT DISTINCT Enrolled.student, dept
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student = MajorsIn.student;
```

- Note: we don't need a `WHERE` clause in this case, because the join condition is now in the `ON` clause.
- Additional selection conditions would go in the `WHERE` clause.

```
<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>25252525</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>english</td>
</tr>
<tr>
<td>66666666</td>
<td>the occult</td>
</tr>
<tr>
<td>33566891</td>
<td>null</td>
</tr>
</tbody>
</table>
```

Outer Joins (cont.)

- Another example: find the IDs and majors of all students enrolled in cscie268 (including those with no major):

```
SELECT Enrolled.student, dept
FROM Enrolled LEFT OUTER JOIN MajorsIn
ON Enrolled.student = MajorsIn.student
WHERE course = 'cscie268';
```

- in this case, there is a `WHERE` clause with an additional selection condition
- the additional condition belongs in the `WHERE` clause because it's not a join condition – i.e., it isn't used to match up tuples from the two tables
Evaluating a `SELECT` command

```sql
SELECT column1, column2, ...
FROM table1, table2, ...
...
```

- The result is equivalent to:
  - evaluating any subqueries in the `FROM` clause
  - forming the Cartesian product of the tables in the `FROM` clause
    ```sql
    table1 x table2 x ...
    ```
  - if there is an OUTER JOIN, applying its join condition and adding extra tuples as needed
  - applying the remaining clauses in the following order:
    ```sql
    WHERE (including any subqueries)
    GROUP BY
    HAVING
    SELECT
    ORDER BY
    ```

CREATE TABLE

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

- **Basic syntax:**
  ```sql
  CREATE TABLE relation_name(
      attribute1_name attribute1_type,
      attribute2_name attribute2_type,
      ...
      attributeN_name attributeN_type
  );
  ```

- **Examples:**
  ```sql
  CREATE TABLE Student(id CHAR(8), name VARCHAR(30));
  CREATE TABLE Room(id CHAR(4), name VARCHAR(30), capacity INTEGER);
  ```
Data Types
• An attribute’s type specifies the domain of the attribute.
• 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.
Terms Used to Express Constraints

- **primary key:**
  CREATE TABLE Student(id char(8) primary key, name varchar(30));
  CREATE TABLE Enrolled(student char(8), course varchar(20), credit_status varchar(15), primary key (student, course));
  - 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 char(4), unique (start_time, end_time, room));
  - 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 char(8), dept varchar(30),
   foreign key (student) references Student(id),
   foreign key (dept) references Department(name));

Student

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

MajorsIn

<table>
<thead>
<tr>
<th>student</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>comp sci</td>
</tr>
<tr>
<td>45678900</td>
<td>mathematics</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Department

<table>
<thead>
<tr>
<th>name</th>
<th>office</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp sci</td>
<td>MD 235</td>
</tr>
<tr>
<td>mathematics</td>
<td>Sci Ctr 520</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Terms Used to Express Constraints (cont.)

- **foreign key / references (cont.):**
  - all values of a foreign key must match the referenced attribute(s) of some tuple in the other relation
  - foreign-key attributes may refer to other attributes in the same relation:
    ```sql
    CREATE TABLE Employee(
        id char(10) primary key, name varchar(30),
        supervisor char(10) references Employee(id);
    ```
  - 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:**
  ```sql
  DROP TABLE  relation_name;
  ```

- **Example:**
  ```sql
  DROP TABLE MajorsIn;
  ```

- **Note:** if a relation is referred to by a foreign key in another relation, it cannot be dropped until either:
  1) the other relation is dropped first
  2) the foreign-key constraint is dropped from the other table
     (we won't look at how to do this)
**INSERT**

- **What it does:** adds a tuple to a relation

- **Syntax:**
  
  ```sql
  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

- **Alternate syntax:**
  
  ```sql
  INSERT INTO relation(attr1, attr2, ...) VALUES (val1, val2, ...);
  ```
  
  - this allows you to specify values of the attributes in a different order, or to specify values for only a subset of the attributes

- If the value of a column is not specified, it is assigned a default value.
  
  - the default value depends on the data type of the column

**INSERT** (cont.)

- **Examples:**
  
  ```sql
  INSERT INTO MajorsIn VALUES ('10005000', 'math');
  ```
  
  [Recall the `CREATE TABLE` command:
   
   ```sql
   CREATE TABLE MajorsIn(student char(8),
   dept varchar(30), ...);
   ```
  
  ```sql
  INSERT INTO MajorsIn(dept, student)
  VALUES ('math', '10005000');
  ```
**INSERT** (cont.)

- The DBMS checks to make sure that an INSERT statement is consistent with the constraints specified for the relation.

- Example: the *MajorsIn* relation was created as follows:
  ```sql
  CREATE TABLE MajorsIn(student char(8),
  dept varchar(30),
  foreign key (student) references Student(id),
  foreign key (dept) references Department(name));
  ```

Given the *Student* relation at left, would the following command be allowed?

```sql
INSERT INTO MajorsIn
VALUES ('98765432', 'math');
```

### Commands for Database Modifications

- **DELETE**: remove one or more tuples from a relation
  - basic syntax:
    ```sql
    DELETE FROM table
    WHERE selection condition;
    ```
  - example:
    ```sql
    DELETE FROM Student
    WHERE id = '10005000';
    ```

---

**Student**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
</tr>
<tr>
<td>25252525</td>
<td>Alan Turing</td>
</tr>
<tr>
<td>33566891</td>
<td>Audrey Chu</td>
</tr>
<tr>
<td>45678900</td>
<td>Jose Delgado</td>
</tr>
<tr>
<td>66666666</td>
<td>Count Dracula</td>
</tr>
</tbody>
</table>

---

CAS CS 460/660, Fall 2013  
David G. Sullivan, Ph.D.  
86
Commands for Database Modifications (cont.)

- **UPDATE**: 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 = 'physics'
    WHERE student = '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 is needed – i.e., do you need to include 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?
Practice Writing Queries

Student(id, name)  Department(name, office)  Room(id, name, capacity)  Course(name, start_time, end_time, room)  MajorsIn(student, dept)  Enrolled(student, course, 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) MajorsIn(student, dept)
Enrolled(student, course, credit_status)

3) Find the number of majors in each department.

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

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

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

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)      MajorsIn(student, dept)
Enrolled(student, course, 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

Computer Science 460/660
Boston University
Fall 2013
David G. Sullivan, Ph.D.

Accessing the Disk

- Relatively speaking, disk I/O is very expensive.
  - in the time it takes to read a 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.
  - pros and cons of each approach?

- 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.)

- Need to consider several different issues:
  - how to map logical records to their physical representation
  - how to organize records on a page
  - how to organize collections of pages
    - called *files*, which may or may not correspond to OS files
    - including the use of index structures
  - what *metadata* is needed and where should it be stored?
    - example: the types and lengths of the fields
    - may need both *per-record* and *per-collection* metadata
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
    - example: in an XML database

- Two options:
  1. fixed-length records: always allocate the maximum possible total length
     - pros and cons?

     | comp | sci |
     |------|-----|
     | math |

  2. variable-length records: only allocate the space needed for a given record
     - pros and cons?

     | comp | sci |
     |------|-----|
     | math |
Format of Fixed-Length Records

- With fixed-length records, the fields can be stored consecutively.
- If a fixed-length record contains a variable-length field, we allocate the max. length and store the actual length.

<table>
<thead>
<tr>
<th>field_1</th>
<th>field_2</th>
<th>field_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567</td>
<td>comp sci #</td>
<td>200</td>
</tr>
<tr>
<td>9876543</td>
<td>math #</td>
<td>125</td>
</tr>
</tbody>
</table>

computing offsets:
- $O_1 = 0$ (zero)
- $O_2 = L_1$
- $O_3 = O_2 + L_2$
- $O_n = O_{n-1} + L_{n-1}$

- To find the position of a field, use the per-collection metadata.
  - if metadata stores the offsets of the fields, use the offset
  - if metadata only stores the field lengths, compute the offset

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.
- In weighing the possible representations, consider:
  - finding/extracting the value of a single field
    ```sql
    SELECT dob
    FROM Person
    ...
    ```
  - updating the value of a single field
    - its length may become smaller or larger
  - adding or removing a field from a collection (e.g., through an ALTER TABLE in SQL)
Format of Variable-Length Records (cont.)

• Options include:
  1. terminate field values by a special delimiter character
     • need to scan from the start of the record to find the value of any field
     • if a field value changes in size, just need to shift the values that come after it (don't need to change their metadata)

  2. precede each field by its length
     • allows us to "jump" over fields while scanning
     • here again, just need to shift values if a field value changes in size

Format of Variable-Length Records (cont.)

• Options (cont.):  

  3. put offsets and other metadata in a record header
     • finding a value?
     • updating a value?
     • what operation requires recomputing offsets in all records?
     • how can we avoid this?
Format of Variable-Length Records (cont.)

- Options (cont.):
  4. store per-record metadata in a separate file
     - key disadvantage?

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 to indicate the presence of a NULL value.
  - 2a: for some fields, a length of 0 could indicate NULL
    - when wouldn't this work?

  - 2b: add additional metadata to the record to track nulls
    - con: waste spaces in collections with few null values

  - 2c: if using an array of offsets, use a special offset
    (e.g., 0 or -1)
Page Format: How to Group Records Together

- Records are grouped together into pages.
- Each record must be assigned a record ID or \textit{rid} that can be used to retrieve it.
  - the \textit{rid} should allow us to find:
    - the page the record is on
    - the location of the record on that page
- We will again consider different schemes for fixed-length and variable-length records.

Page Format with Fixed-Length Records

- Can use a fairly simple scheme:
  - let \( s \) = the size of each record
  - the maximum \# of records per page is given by the \textit{fill factor}, \( f = \text{page\_size} / s \)
  - \textbf{ex}: 4K pages, 512-byte records
    \[ f = \frac{4096}{512} = 8 \]
  - to find the record whose \textit{rid} = \( i \):
    - page \#: \( p = i / f \)
    - record \# on page: \( r = i \mod f \)
    - offset on page: \( o = r \times s \)
  - \textbf{ex}: where is record 13?

<table>
<thead>
<tr>
<th>page</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

- Can we move a record to a different location on the same page?
Page Format with Fixed-Length Records (cont.)

- To deal with deletions and insertions:
  - add a bit to each record indicating whether it is deleted
    - so that it won't be visited during scans of the records
  - use a free list: store the location of the first empty record in a special header page; each empty record points to the next one

Page Format with Variable-Length Records

- Why does the fixed-length scheme break down?
  -
  -

- One possible approach:
  - let rid = (page #, offset within page)
  - problem: can't move a record without changing its rid
  - reasons for moving a record within a page:
    - when an update changes a record's size
      - example: what if record 0 is updated and is now 100 bytes?

  - to make room for other records
    - example: record 0: 70 → 50 bytes
      - record 1: 80 → 60 bytes
      - we have 40 free bytes, but can't add records larger than 20 unless we move rec 1
Page Format with Variable-Length Records (cont.)

- A better approach:
  - add a header to each page that includes a table of offsets
  - \( \text{rid} = (\text{page} \#, \text{index into offset table}) \)
  - when we move a record on the page, we change its offset but we don’t change its rid!
  
    | example: |
    |---|
    | page 10: |
    | 0 1 2 |
    | (10, 0) |
    | (10, 1) |
    | (10, 2) |
    | record (10, 1) grows in size |
    | page 10: |
    | 0 1 2 |
    | (10, 0) |
    | (10, 1) |
    | (10, 2) |
    | (10, 1) |
    | (10, 2) |

- Problem: how large an offset table should we preallocate?
- Solution: don’t preallocate!
  - start adding records at the end of the page and work backwards toward the beginning
  - doing so allows us to grow the offset table as needed

Other Issues

- If a record becomes too large to fit on a page, we can move it to another page and have its offset-table entry store the new location.
  
    | Other Issues |
    |---|
    | page 10: |
    | 0 1 2 |
    | (10, 0) |
    | (10, 1) |
    | (10, 2) |
    | (10, 0) |
    | (10, 1) |
    | (10, 2) |
    | (10, 2) |
    | page 10: |
    | 0 1 2 |
    | (10, 0) |
    | (10, 1) |
    | (10, 2) |
    | (10, 0) |
    | (10, 1) |
    | (10, 2) |
    | (10, 2) |

- If a record is too large to fit on any page, we can put it in one or more special pages called \textit{overflow pages}.

- If a record is deleted, we put a special value in its offset-table entry so that scans will not try to access it.

- The page header needs to include info. that can be used to determine where the free space is on the page.
Database Files

- A logical collection of database pages is referred to as a file.
  - may or may not correspond to an OS file
- The DBMS needs to manage the pages within a given file:
  - keeping track of all used pages, to support scans of all records
  - keeping track of unused pages and/or used pages with some free space, to support efficient insertions

Index Structures

- Index structures allow for more efficient access to data records.
- An index structure stores (key, data) pairs.
  - provides efficient inserts and searches based on the key
- We'll look at two examples of index structures:
  - B-trees
  - hash tables
- First, we need to consider two ways that an index structure can be used in a database.
Clustered/Primary Index

- A given collection of data records is typically stored within an index structure.
  - a clustered index or internal index
  - a primary index, because it is typically based on the primary key

- If records are stored randomly (outside 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 additional indices based on other fields.
  - an unclustered or external index
  - a secondary index

- Example: `Customer(id, name, street, city, state, zip)`
  - primary index: key = id, data = rest of the record
  - secondary index: key = name, data = id

- Need two lookups when using a secondary index:

<table>
<thead>
<tr>
<th>secondary index value</th>
<th>secondary primary key</th>
<th>primary full record</th>
</tr>
</thead>
</table>
  | 'Ted Codd'            | '012345'              | ('012345', 'Ted Codd',…))
B-Trees

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

- Example: a B-tree of order 2

  ![B-tree example](image)

  (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

- Algorithm for searching for an entry by key:

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

- Example: search for the entry whose key is 87

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

Insertion in B-Trees

- Algorithm for inserting an entry with a key k:
  search for k, which will bring you to a leaf node (ignoring duplicates for now)
  if the leaf node has fewer than 2m entries, add the new entry to the leaf node
  else split the node, dividing up the 2m + 1 entries:
    the first/smallest m entries remain
    the last/largest m entries go in a new node
    send the middle entry up and insert it (and a pointer to the new node) in the parent

- Example of an insertion without a split: insert 13
  \( m = 2 \)
Splits in B-Trees

- Insert 5 into the result of the previous insertion:
  - The middle item (the 10) was sent up to the root. It has no room, so it is split as well, and a new root is formed:
  - Splitting the root increases the tree’s height by 1, but the tree is still balanced. This is only way that the tree’s height increases.
  - When an internal node is split, its $2m + 2$ 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 a record involves traversing a single path from the root to a leaf node.
  - # of nodes accessed $\leq$ tree height + 1
  - each internal node has at least $m$ children $\rightarrow$ tree height $\leq \log_m n$, where $n =$ # of entries
  - search is $O(\log_m n)$
  - insertion is also $O(\log_m n)$
- To minimize disk I/O, make $m$ as large as possible.
  - if 1 million records: access at most 10 pages when $m = 4$
  - access at most 3 pages when $m = 32$
- The fanout of a tree is the average number of children per node.
  - if fanout = $f$, tree height $= O(\log_f n)$
B+Trees

• A B+tree is a B-tree variant in which:
  • data entries are only found in the leaf nodes
  • internal nodes contain only keys and child pointers
  • an entry’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 entries are in leaf nodes, we can link the leaves together to improve the efficiency of operations that involve scanning the entries in key order (e.g., range searches)
Differences in the Algorithms for B+Trees

• When searching, we don’t stop if the key is encountered in an internal node. Instead, we keep going until we reach a leaf node.

• When splitting a leaf node with $2m + 1$ entries:
  • the first $m$ entries remain in the original node as before
  • all of the remaining $m + 1$ entries are put in the new node, including the middle entry
  • the key of the middle entry is copied into the parent
    • why can't we move it up?

• Example: insert 18

  $m = 2$

  ![Diagram showing the insertion process](image)

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 entry and remove it.

• If a node N ends up with fewer than \( m \) items, do one of the following:
  • take one or more items from a sibling node with more than \( m \) items and add them to N
  • if the sibling node only has \( m \) items, merge N with the sibling
    • reverses the process involved in splitting a node

• If the key of the removed entry is in an internal node, don’t remove it from the internal node, since we need the key to navigate to the node’s children.
  • can remove it if the associated child node is eventually merged with a sibling

• Some systems don’t worry about nodes with too few items, since they assume items will be added again eventually.

Ideal Case: Searching = Indexing

• The optimal search and insertion performance would be achieved if we could create an index structure in which the following condition held:
  • key of data item = page on which the item is located

• In most real-world problems, we can’t do this.
  • the key values may not be integers
  • the range of possible key values may be so large that we can’t afford to give each key value its own page

• To handle these problems, we perform hashing:
  • use a hash function to convert the keys to page numbers

• 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
  • need to deal with collisions = two values hashed to same index

• On-disk:
  • the hash value tells you which page the entry should be on
  • because a page is fairly large, each page serves as a bucket that stores multiple entries
  • need to deal with full buckets

Dealing with Full Buckets

• Two approaches:
  1) chaining: link one or more overflow buckets off the primary one

  
  ![Diagram of chaining]

  2) open addressing: find another bucket that is not full
  • similar to open addressing in on-memory hash tables
Static vs. Dynamic Hashing

- In static hashing, the number of buckets never changes.
  - problem: databases typically grow over time
    - preallocating a large number of buckets from the start can be wasteful in the short term
    - if the number of buckets is too small, we end up with large chains of overflow buckets

- In dynamic hashing, the number of buckets can change over time.
  - can be expensive if you're not careful
  - linear hashing tries to minimize the work involved in growing the table

A Simplistic Approach to Dynamic Hashing

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

- Example:
  - h(key) = number of characters in key
  - Use modulus operator to determine where to put a key:
    
    \[
    \text{bucket index} = h(\text{key}) \mod \text{number of buckets}
    \]

    
    0  "if", "case", "continue"

    1  "class", "for", "extends"

    0
    1
    2
    3
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:
  \[
i = \text{round}(\log_2 n) \quad \text{where } n \text{ is the current number of buckets}
  \]
  - example: \( n = 3 \rightarrow i = \text{round}(\log_2 3) = 2 \)
- If there’s a bucket with the index given by the \( i \) rightmost bits, put the key there.

\[
\begin{array}{l}
  h(\text{"if"}) = 2 = 00000010 \quad 00 = 0 \quad \text{\"case", \"continue\"} \\
  h(\text{"case"}) = 4 = 00000100 \quad 01 = 1 \quad \text{\"class", \"for", \"extends\"} \\
  h(\text{"class"}) = 5 = 00000101 \quad 10 = 2 \quad \text{\"if"} \\
  h(\text{"continue"}) = 8 = 00001000 \quad 11 = 3 \text{ is too big, so use 1}
\end{array}
\]
- If not, use the bucket specified by the rightmost \( i - 1 \) bits

\[
\begin{array}{l}
  h(\text{"for"}) = 3 = 00000011 \quad (11 = 3 \text{ is too big, so use 1}) \\
  h(\text{"extends"}) = 7 = 00000111
\end{array}
\]

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 \( 1 \text{xyz} \) (xyz = binary digits), rehash the bucket with binary index \( 0 \text{xyz} \)
- 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 \( \rightarrow i = \text{round}(\log_2 4) = 2 \) (unchanged)

```
<table>
<thead>
<tr>
<th>n = 3, i = 2</th>
<th>n = 4, i = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 = 0</td>
<td>00 = 0</td>
</tr>
<tr>
<td>&quot;case&quot;, &quot;continue&quot;</td>
<td>&quot;case&quot;, &quot;continue&quot;</td>
</tr>
<tr>
<td>01 = 1</td>
<td>01 = 1</td>
</tr>
<tr>
<td>&quot;class&quot;, &quot;for&quot;, &quot;extends&quot;</td>
<td>&quot;class&quot;, &quot;for&quot;, &quot;extends&quot;</td>
</tr>
<tr>
<td>10 = 2</td>
<td>10 = 2</td>
</tr>
<tr>
<td>&quot;if&quot;, &quot;switch&quot;</td>
<td>&quot;if&quot;, &quot;switch&quot;</td>
</tr>
<tr>
<td>11 = 3</td>
<td>11 = 3</td>
</tr>
</tbody>
</table>
```

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:
    - \( h(\text{"class"}) = 5 = 00000101 \)
    - \( h(\text{"for"}) = 3 = 00000011 \) (move to new bucket)
    - \( h(\text{"extends"}) = 7 = 00000111 \) (move to new bucket)

```
<table>
<thead>
<tr>
<th>n = 4, i = 2</th>
<th>n = 4, i = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 = 0</td>
<td>00 = 0</td>
</tr>
<tr>
<td>&quot;case&quot;, &quot;continue&quot;</td>
<td>&quot;case&quot;, &quot;continue&quot;</td>
</tr>
<tr>
<td>01 = 1</td>
<td>01 = 1</td>
</tr>
<tr>
<td>&quot;class&quot;, &quot;for&quot;, &quot;extends&quot;</td>
<td>&quot;class&quot;</td>
</tr>
<tr>
<td>10 = 2</td>
<td>10 = 2</td>
</tr>
<tr>
<td>&quot;if&quot;, &quot;switch&quot;</td>
<td>&quot;if&quot;, &quot;switch&quot;</td>
</tr>
<tr>
<td>11 = 3</td>
<td>11 = 3</td>
</tr>
<tr>
<td>&quot;for&quot;, &quot;extends&quot;</td>
<td></td>
</tr>
</tbody>
</table>
```
Additional Details

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

\[
\begin{array}{|c|c|}
\hline
n = 4, i = 2, f = 9, 9 > 2^4 & n = 5, i = 3 \\
\hline
00 = 0 & \text{"case"}, \text{"continue"} \\
01 = 1 & \text{"class"}, \text{"while"} \\
10 = 2 & \text{"if"}, \text{"switch"}, \text{"String"} \\
11 = 3 & \text{"for"}, \text{"extends"} \\
\hline
\end{array}
\]

- 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 entries 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 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)

\[
\begin{array}{|c|c|}
\hline
\text{"toString"}: h("toString") & ? \\
\text{"private"}: h("private") & ? \\
\text{"interface"}: h("interface") & ? \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
n = 5, i = 3 & n = 6, i = 3 \\
\hline
000 = 0 & \text{"continue"} \\
001 = 1 & \text{"class"}, \text{"while"} \\
010 = 2 & \text{"if"}, \text{"switch"}, \text{"String"} \\
011 = 3 & \text{"for"}, \text{"extends"} \\
100 = 4 & \text{"case"} \\
101 = 5 & \\
\hline
\end{array}
\]
Hash Table Efficiency

- Assume that we're using chaining.
- 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.
Hash Table vs. B-Tree/B+tree

- For on-disk data, a B-tree / B+tree is typically better.
  - internal nodes typically fit in the cache, so we only go to disk for the leaf node
    - same # of disk accesses as a hash table
  - once a leaf page is in the cache, it’s often the case that other items on it will be accessed soon because the items are sorted by key
    - no disk access needed in such cases
  - efficiently supports a wider range of queries – see previous page

- Use a hash table if the access pattern is fairly random or if the database is so large that the cache doesn’t help.
Implementing a Logical-to-Physical Mapping

Computer Science 460/660
Boston University
Fall 2013
David G. Sullivan, Ph.D.

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 will give 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 and returns a reference to an object that is an instance of a subclass of the class `SQLStatement`.

- `SQLStatement` is abstract.
  - contains fields and methods inherited by the subclasses
  - includes an abstract `execute()` method

- You'll implement the `execute()` method for some of the subclasses.
SQLStatement Class

- Looks something like this:

```java
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 all methods static, so that 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
  - putMetadata(), getMetadata(), removeMetadata()
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.

• You will use the Berkeley DB Java API.
  • not Berkeley DB Java Edition (JE).

• We are also 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"

• Possible index structures: btree, hash, record-number, queue
  • referred to in the BDB documentation as access methods

• 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 (caching, logging, transactions, locking) for those databases

- RDBMS: related tables are grouped together into a database. BDB: related databases are grouped together into an environment.

- Typically, the files for the BDB databases associated with a given environment are put in the same directory.
  - known as the environment’s home directory.

- An environment is operated on by making method/function calls using an environment handle – an instance of the Environment class.

Opening/Creating a BDB Database

- The environment must be configured and opened first.
  - we’ll give you the code for this

- Then you open/create the database.
  - create an object that specifies the database’s configuration:
    
    ```java
    DatabaseConfig config = new DatabaseConfig();
    config.setType(DatabaseType.BTREE);
    config.setAllowCreate(true);
    ...
    ```

  - use the environment handle to open the database, passing the DatabaseConfig object as an argument:
    
    ```java
    Environment dbenv = DBMS.getEnv();
    Database db = dbenv.openDatabase(null, "movie.db", null, config);
    ```
Key/Value Pairs

- In Berkeley DB, the on-disk keys and values are *byte arrays* – i.e., collections of bytes.

- Berkeley DB does *not* attempt to interpret them.

- Your code will need to impose structure on these byte arrays.
  - see the earlier notes on record formats, and the later notes on marshalling data

Key/Value Pairs (cont.)

- When manipulating keys and values within a program, we represent them using a `DatabaseEntry` object.

- For a given key/value pair, we need two `DatabaseEntry` s.
  - one for the key, and 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`, ...
Inserting Data into a BDB Database

- Create the DatabaseEntry objects for the key and value:
  ```java
  String keyStr = "cscie268";
  String valueStr = "Maxwell Dworkin G-135";
  DatabaseEntry key =
      new DatabaseEntry(keyStr.getBytes("UTF-8"));
  DatabaseEntry value =
      new DatabaseEntry(valueStr.getBytes("UTF-8"));
  ```
  - if we didn't specify UTF-8, the JVM would use the default encoding, and the default encoding could subsequently change.

- Use the Database handle's put method:
  ```java
  Database db; // assume it has been opened
  OperationStatus ret = db.put(null, key, value);
  ```
  - the first argument to put can be used to specify the transaction in which the put is occurring
  - null indicates no transaction

Retrieving a Single Key/Value Pair

- Create the DatabaseEntry objects for the key and value:
  ```java
  String keyStr = "cscie268";
  DatabaseEntry key =
      new DatabaseEntry(keyStr.getBytes("UTF-8"));
  DatabaseEntry value = new DatabaseEntry(); // none yet
  ```

- Use the Database handle's get method:
  ```java
  Database db; // assume it has been opened
  OperationStatus ret = db.get(null, key, value, null);
  if (ret == OperationStatus.NOTFOUND) {
      System.err.println("no item with this key");
  } else if (ret == OperationStatus.SUCCESS) {
      byte[] valueBytes = value.getData();
      String valueStr = new String(valueBytes, "UTF-8");
      System.out.println(keyStr + " meets in " + valueStr);
  }
  ```
Cursors in Berkeley DB

- In general, a cursor is a construct used to iterate over the records in a database file.
- In BDB, a cursor iterates over key/value pairs in a BDB database.
- Cursor operations are performed by making method/function calls using a cursor handle (an object or struct).
  - an instance of the Cursor class

Opening a Cursor

- Use the Database handle's openCursor() method:
  ```java
  Database db;  // assume it has been opened
  Cursor curs = db.openCursor(null, null);
  ```
- A cursor that has just been opened is not yet pointing to any record.
  - attempts to get the current record will fail
- A cursor is initialized by performing a method/function call to get one of the records.
if a cursor is not yet initialized, attempts to get the next record will retrieve the first record
• after a delete, the position of the cursor doesn’t change, so you need to reposition it before trying to do something with the current record

<table>
<thead>
<tr>
<th>operation</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>get the first record</td>
<td>getFirst()</td>
</tr>
<tr>
<td>get the record with the specified key</td>
<td>getSearchKey()</td>
</tr>
<tr>
<td>get the next record</td>
<td>getNext()</td>
</tr>
<tr>
<td>get the current record</td>
<td>getCurrent()</td>
</tr>
<tr>
<td>modify the current record</td>
<td>putCurrent()</td>
</tr>
<tr>
<td>delete current record</td>
<td>delete()</td>
</tr>
<tr>
<td>close the cursor</td>
<td>close()</td>
</tr>
</tbody>
</table>

• a cursor is not yet initialized, attempts to get the next record will retrieve the first record
• after a delete, the position of the cursor doesn’t change, so you need to reposition it before trying to do something with the current record

Iterating Over All Records Using a Cursor

• The key/value pairs are returned in DatabaseEntry that are passed as parameters to the getter method.
• Assume we have an open cursor whose handle is curs.
• Here’s an example:

```java
DatabaseEntry key = new DatabaseEntry();
DatabaseEntry value = new DatabaseEntry();
while (curs.getNext(key, value, null) == OperationStatus.SUCCESS) {
    String keyStr = new String(key.getData(), "UTF-8");
    String valueStr = new String(value.getData(), "UTF-8");
    System.out.println(keyStr + " meets in " + valueStr);
}
```
Marshalling Data

- We need to be able to take a collection of fields and store them in a key/value pair – each component of which is a byte array
  - example:
    
    
    
    | '1234567' | 'comp sci' | 200 |
    |-----------|------------|-----|
    | 16 24 28   | comp sci   | 200 |

- This process is referred to as *marshalling* the data.
- The reverse process is known as *unmarshalling*.

Marshalling Data with BDB’s Java API

- In theory, we could use Java serialization to convert objects to byte arrays and back.
  - produces unnecessarily large records, because class information is stored in each record!
  - it’s also slow, and you can’t sort the resulting byte arrays
- Instead, we'll make use of some of the classes from BDB’s *Bind API*.
  - provide methods for storing various types of data into byte arrays, and vice versa
Classes from the BDB Bind API

- **TupleOutput**: an output stream with methods that write values into a byte array (similar to Java's `DataOutputStream`)
  - `writeByte(int val)`
  - `writeBytes(String val)`
  - `writeInt(int val)`
  - `writeDouble(double val)`

- **TupleInput**: an input stream with methods that read values from a byte array (similar to Java's `DataInputStream`)
  - `readByte()`
  - `readBytes(int length)`
  - `readInt()`
  - `readDouble()`

- We'll give you more info about using the Bind API in the assignments.
Overview

- Steps in processing a relational query:
  1) parse the SQL command to produce a parse tree
  2) create an initial query plan
     - specifies the operations from relational algebra that are needed to evaluate the query
  3) convert the initial plan to an equivalent optimized query plan
     - this is still a logical plan, expressed in terms of relational algebra, but it is expected to require less time
  4) generate a physical query plan for carrying out the optimized logical query plan
     - map each relational-algebra operation to an algorithm that accesses/manipulates the data on disk
  5) execute the physical query plan
Example of a Logical Query Plan

- Relations: Person(id, name)
  Enrolled(student_id, course_name, credit_status)

- Query: find the names of all students enrolled in CS 165

  ```sql
  SELECT name
  FROM Person, Enrolled
  WHERE id = student_id
  AND course_name = "cs165";
  ```

- Initial query plan:

  \[ \pi_{\text{name}} (\sigma_{\text{id} = \text{student_id} \land \text{course_name} = \"cs165\"}(\text{Person} \times \text{Enrolled})) \]

  in other words: cross Person and Enrolled
  select tuples that match the predicate
  project out the name attribute

Optimizing a Logical Query Plan

\[ \pi_{\text{name}} (\sigma_{\text{id} = \text{student_id} \land \text{course_name} = \"cs165\"}(\text{Person} \times \text{Enrolled})) \]

\[ \pi_{\text{name}} (\sigma_{\text{course_name} = \"cs165\"}(\text{Person} \bowtie_{\text{id} = \text{student_id}} \text{Enrolled})) \]

\[ \pi_{\text{name}} (\text{Person} \bowtie_{\text{id} = \text{student_id}} (\sigma_{\text{course_name} = \"cs165\"}(\text{Enrolled}))) \]

- These optimizations are performed by a query optimizer.
  - very complicated subsystem of the DBMS
  - subject of lots of research
  - can get in the way if your query is simple
Review: Index Structures

• In a *clustered index* or *internal index*, the data records are stored within the index itself.
  • sometimes called a *primary index*

• In an *unclustered index* or *external index*, the data records are stored in another file, and the index stores (key, record ID) pairs.
  • sometimes called a *secondary index*

Physical Query-Plan Operations

• Many of the operations in a physical query plan are algorithms for carrying out a relational-algebra operation

• In addition, there are other basic physical operations that you need, such as:
  • scanning through a relation
  • scanning through a sorted version of a relation
    • sorting it as needed
      ▪ usually involves merging sorted subrelations
      ▪ similar to mergesort, but optimized to reduce disk I/O
Using Iterators

• Physical query-plan operations are implemented using *iterators*.

• Allows us to gradually access the results of an operation – one tuple at a time.
  - example: a *selection iterator* that gets the tuples that satisfy a selection condition
  - have typical iterator methods: first(), next(), etc.

• Using an iterator:
  - is generally preferable to *materializing* the entire result of an operation
    - i.e., creating a temporary relation for it
    - allows us to combine operations more efficiently

• What BDB construct would you use to implement an iterator?

Using Iterators (cont.)

• Iterator methods can include:
  - open(): construct the iterator, opening the underlying resources and performing whatever initial steps are required
  - first(): position the iterator on the first tuple in the result
  - next(): position the iterator on the next tuple in the result
  - getTuple(), getColumnVal(): access the tuple on which the iterator is positioned
  - close(): free/close the underlying resources

• first() and next() return true on success and false on failure
  - first(): fails if no tuple satisfies the result
  - next(): fails if there are no remaining tuples that satisfy the result
  - next() on a newly opened iterator acts like first()

• How is an iterator different from a cursor?
Using Iterators: An Example

• Query plan: $\pi_{\text{name}} (\sigma_{\text{course_name} = 'cs165'} (\text{Person} \times_{\text{id} = \text{student_id}} \text{Enrolled}))$

• Without an iterator:
  apply the join $\rightarrow$ the complete, materialized join (call it $J$)
  apply the selection to the join $\rightarrow$ materialized relation $S$
  apply the projection to $S$ $\rightarrow$ final result

• With iterators, we can effectively do this:
  while there are unseen combinations of tuples from Person and Enrolled:
    advance to the next combination satisfying the join
    if that combination satisfies the selection predicate
    perform the projection on it and print it

Composing Iterators

• One iterator may make use of another iterator.

• Example: $\pi_{\text{name}} (\sigma_{\text{course_name} = 'cs165'} (\text{Person} \times_{\text{id} = \text{student_id}} \text{Enrolled}))$

  • construct table-scan iterators for Person ($I_P$) and Enrolled ($I_E$)
  • construct a join iterator ($I_J$) that uses $I_P$ and $I_E$ to get tuples from the two tables
    • its next() would call $I_P$ and $I_E$'s next() and first() until it got a combination that satisfied the join condition
  • construct a $\sigma$ iterator ($I_\sigma$) that uses $I_J$ to get tuples from the join
  • construct a $\pi$ iterator ($I_\pi$) that uses to $I_\sigma$ to get the tuples from which it should project
Iterators in the DBMS Code Framework

- There are partial implementations of three types of iterators:
  - **a table iterator**: iterates over the tuples in:
    - a entire single table
    - or the relation that is produced by applying a selection operator to the tuples of single table
  - **a cross-product iterator**: iterates over the tuples in:
    - the entire cross product of two or more tables
    - or the relation that is produced by applying a selection operator to the cross product
    - it uses table iterators to get the tuples from the tables being crossed!
  - **a projection iterator**: iterates over the tuples in the relation formed by applying a predicate operator to another relation.
    - it uses either a table or cross-product iterator!

Selection on a Single Table

- Examples:
  - assume this table: \textit{Employee}(id, name, salary)
  - how would you perform the following selection query in BDB?
    
    ```sql
    SELECT *
    FROM Employee
    WHERE id = '12345678';
    ```

  - what about this one?
    
    ```sql
    SELECT *
    FROM Employee
    WHERE name = 'Dave Sullivan';
    ```
Selection on a Single Table (cont.)

- Case 1: single equality predicate (e.g., \(id = "12345678"\))
  - if primary index is based on the attribute in the predicate: use it to get the tuple or tuples that satisfy the predicate
    - B-tree: access at most \(O(h)\) pages, \(h = \text{tree height}\)
  - else if secondary index on the attribute: use it unless the number of matching tuples will be large
    - access \(O(r)\) pages, \(r = \# \text{ of matching records}\)
    - why might this be worse than a scan if \(r\) is large?
  - else scan the table and apply the predicate to all tuples

Selection on a Single Table (cont.)

- Case 2: single predicate with a comparison (e.g., \(\text{runtime} < 120\))
  - primary index sorted on the attribute in the predicate:
    - \(>\) or \(>=\): ?
    - \(<\) or \(<=\): ?
  - secondary index: only if selecting a small number of records
  - else scan the table

- Case 3: compound predicates with AND
  - if there is an appropriate* index for one of the simple predicates, use it to get candidate tuples and then apply the other predicate(s)
  - else scan the table

* appropriate = would be used if just the simple predicate were being evaluated
Selection on a Single Table (cont.)

- Case 4: compound predicates with OR
  - if there are appropriate indices for all of the simple predicates, use them and output the union of the results
    - how could we produce the union?
  - else scan the table

Nested-Loop Join

- The basic algorithm for joining two relations is called \textit{nested-loop join}.

- Algorithm for \( R \bowtie_\theta S \), where \( \theta \) is the predicate:
  
  for each tuple \( r \) in \( R \) {
    for each tuple \( s \) in \( S \) {
      if \((r, s)\) satisfies \( \theta \), add it to the result
    }
  }


Nested-Loop Join (cont.)

- Here's the algorithm using table-scan iterators:
  \[ I_R = \text{an iterator that scans } R \text{ (open/construct it)} \]
  \[ I_S = \text{an iterator that scans } S \text{ (open/construct it)} \]

  \[
  \text{while (}I_R\text{.next())} \{
  \text{if (!}I_S\text{.first())}
  \text{return}
  \text{do {}
  \text{if (}I_R\text{.getTuple()}, }I_S\text{.getTuple()) satisfies } \theta,
  \text{add it to the result}
  \text{}} \text{while (}I_S\text{.next())}
  \}
  \]

  \[ I_R\text{.close()}
  \]

  \[ I_S\text{.close()}
  \]

An Iterator for Nested-Loop Join

- What if we want a nested-loop-join iterator for \( R \bowtie \_ \theta S \)?
  - gives us one tuple at a time from the result of the join
  - build it using table-scan iterators for \( R \) and \( S \)

- Its \text{next()} method might look something like this:

  \[
  \text{if (count of tuples visited == 0) } \{
  \text{if (!}I_R\text{.first())}
  \text{return false;}
  \}
  \]

  \[
  \text{do {}
  \text{if (!}I_S\text{.next())} \{
  \text{if (!}I_S\text{.first())}
  \text{return false;}
  \text{if (!}I_R\text{.next())}
  \text{return false;}
  \}
  \} \text{while (}I_R\text{.getTuple()}, }I_S\text{.getTuple()) does not satisfy } \theta
  \]

  increment count of tuples visited
  return true
Example of Nested-Loop Join

CREATE TABLE Employee (id CHAR(8) PRIMARY KEY, name CHAR(20), dept CHAR(10), salary REAL, FOREIGN KEY (dept) REFERENCES Department(name));
CREATE TABLE Department (name CHAR(10) PRIMARY KEY, manager CHAR(20));

• Query: SELECT Employee.name, Department.manager FROM Employee, Department WHERE Employee.dept = Department.name;

• Assume that:
  100,000 employees; Employee relation occupies 1000 blocks
  100 departments; Department relation occupies 10 blocks
  the database cache can store 500 blocks

• How many disk reads will the nested-loop join perform?

Block Nested-Loop Join

• Nested-loop join is very inefficient.
  • each block in S is read in from disk once for every tuple in R

• Block nested-loop join is more efficient.

• Algorithm for the theta join $R \bowtie_{\theta} S$, where $\theta$ is the predicate:
  for each block $B_R$ in $R$ {
    for each block $B_S$ in $S$ {
      for each tuple $r$ in $B_R$ {
        for each tuple $s$ in $B_S$ {
          if $(r, s)$ satisfies $\theta$, add it to the result
        }
      }
    }
  }

• Now each block in S is read in once for each block in $R$, rather than once for each tuple in $R$. 
Index Nested-Loop Join

- When:
  - the join involves an equality condition (e.g., R.foo = S.bar)
  - there is an appropriate* index on the relation in the inner loop
    we can replace the inner loop with a selection that uses the index

- Example: \[R \bowtie_{R.foo = S.bar} S\]

  for each tuple \(r\) in \(R\) {
    use index to "jump" to the first tuple \(s\) in \(S\) which \(s.bar = r.foo\)
    for each such tuple \(s\) {
      add \((r, s)\) to the result
    }
  }

* appropriate = an index that would be used for a simple selection query
  involving \(S.bar\) (see the earlier slides)

Our Example Revisited

CREATE TABLE Employee (id CHAR(8) PRIMARY KEY,
  name CHAR(20), dept CHAR(10), salary REAL,
  FOREIGN KEY (dept) REFERENCES Department(name));

CREATE TABLE Department (name CHAR(10) PRIMARY KEY,
  manager CHAR(20));

- Query:  
  SELECT Employee.name, Department.manager
  FROM Employee, Department
  WHERE Employee.dept = Department.name;

- How many combinations of tuples would standard nested-loop
  join consider (100,000 employees, 100 departments)?

- Can we use index nested-loop join?
- If so, how many combinations of tuples would it consider?
- Would it decrease the number of disk reads in this case?
Hash Join

- Can only be used for joins with equality conditions

- Basic idea:
  - create two hash tables, one for the tuples in R and one for the tuples in S
  - hash tuples into buckets on the basis of the join attributes
  - match up tuples in a given bucket of R's hash table with tuples in the corresponding bucket of S's hash table

- Another variant if one of the two relations is relatively small:
  - only hash the smaller relation S, such that its hash table fits in memory
  - iterate over the larger relation L, and perform a single hash lookup for each tuple in L
  - why wouldn't this work well if both relations were large?

Implementing Projection

- Seems simple: just extract the specified attributes

- When would it be more complicated?

- How could you implement that case?
Implementing CREATE TABLE

- Examples:

  CREATE TABLE Person(id CHAR(8) PRIMARY KEY, 
                      name VARCHAR(30));

  CREATE TABLE Room(id CHAR(4) PRIMARY KEY, 
                    name VARCHAR(20), capacity INT);

  CREATE TABLE Course(name CHAR(8) PRIMARY KEY, 
                      start TIME, end TIME, room_id CHAR(4), 
                      UNIQUE (start, end, room_id), 
                      FOREIGN KEY (room_id) REFERENCES Room(id));

  CREATE TABLE Enrolled(student_id char(8), 
                         course_name char(8), credit_status char(4), 
                         foreign key (student_id) references Person(id), 
                         foreign key (dept_id) references Department(id));

- What steps would the DBMS need to take?

Implementing INSERT

- Examples:

  INSERT INTO Person VALUES("12345678", "Alan Turing");

  INSERT INTO Enrolled VALUES("12345678", 
                            "CS 165", "ungr");

- What steps would the DBMS need to take?
Implementing DELETE

• Example:
  
  ```sql
  DELETE FROM Person WHERE id = "12345678"
  ```

• What steps would the DBMS need to take?

Implementing UPDATE

• Example:
  
  ```sql
  UPDATE Person
  SET name = "A. Turing"
  WHERE id = '12345678'
  ```

• What steps would the DBMS need to take?
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.
   • in SQL: \texttt{BEGIN WORK} or \texttt{START TRANSACTION}

2. Perform the operations in the transaction.
   • in SQL: a set of SQL commands (\texttt{SELECT}, \texttt{UPDATE}, etc.)

3. End the transaction in one of two ways:
   • \textit{commit} it: make all of its results visible and persistent
     • all of the changes happen
     • in SQL: \texttt{COMMIT WORK} or \texttt{COMMIT}
   • \textit{abort} it: undo all of its changes, returning the system to the state it was in before the transaction began
     • none of the changes happen
     • in SQL: \texttt{ROLLBACK WORK} or \texttt{ROLLBACK}

Why Do We Need Transactions?

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

\begin{verbatim}
read balance1
write(balance1 - 500)
CRASH
read balance2
write(balance2 + 500)
\end{verbatim}
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

  ```plaintext
  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)
  read balance2
  write(balance2 + 500)
  ```

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 expressed in 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 completes, its changes survive failures

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

- The DBMS guarantees atomicity, isolation, and durability.
Atomicity and Durability

- These properties are guaranteed by the part of the system that performs logging and recovery.

- As transactions execute, key details of their execution are recorded in a log file.

- After a crash, the system performs recovery by using the log to:
  - *redo* changes by any txn that committed before the crash
    - durability: the txn’s changes are still there after the crash
    - atomicity: all of its changes take effect
    - why weren’t all of the changes on disk before recovery?

  - *undo* changes made by any txn that didn’t commit
    - atomicity: none of its changes take effect

- 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:

  \[
  \text{transaction } T_1 \\
  \begin{align*}
  \text{read } & \text{balance}_1 \\
  \text{write} & \text{(balance}_1 - 500) \\
  \text{read } & \text{balance}_2 \\
  \text{write} & \text{(balance}_2 + 500)
  \end{align*}
  \]

  \[
  \text{transaction } T_2 \\
  \begin{align*}
  \text{read } & \text{balance}_1 \\
  \text{read } & \text{balance}_2 \\
  \text{if } & (\text{balance}_1 + \text{balance}_2 < \min) \\
  \text{write} & \text{(balance}_1 - \text{fee)}
  \end{align*}
  \]

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

  \[
  \begin{align*}
  \text{read } & \text{balance}_1 \\
  \text{read } & \text{balance}_2 \\
  \text{if } & (\text{balance}_1 + \text{balance}_2 < \min) \\
  \text{write} & \text{(balance}_1 - \text{fee)}
  \end{align*}
  \]

  or

  \[
  \begin{align*}
  \text{read } & \text{balance}_1 \\
  \text{write} & \text{(balance}_1 - 500) \\
  \text{read } & \text{balance}_2 \\
  \text{read } & \text{balance}_2 \\
  \text{write} & \text{(balance}_1 + \text{balance}_2 + 500)
  \end{align*}
  \]

- doesn’t make sense for performance reasons. why?
Serializability

- Rather than using a serial schedule, it is sufficient to use a **serializable schedule** – one whose effects are equivalent to the effects of some serial schedule.

- Example:

  T1: 
  - `read balance1`
  - `write(balance1 - 500)`
  - `read balance2`
  - `write(balance2 + 500)`
  - `read balance1`
  - `read balance2`
  - `if (balance1 + balance2 < min)`
  - `write(balance1 - fee)`

- T2: 
  - `read balance1`
  - `read balance2`
  - `write(balance2 + 500)`
  - `write(balance1 - 500)`

- this is equivalent to which serial schedule?

Serializability (cont.)

- Is the following schedule serializable?

  T1: 
  - `read balance2`
  - `write(balance2 + 500)`
  - `read balance1`
  - `write(balance1 - 500)`
  - `read balance1`
  - `read balance2`
  - `if (balance1 + balance2 < min)`
  - `write(balance1 - fee)`

  T2: 
  - `read balance1`
  - `read balance2`
  - `write(balance2 + 500)`
  - `write(balance1 - 500)`
  - `if (balance1 + balance2 < min)`
  - `write(balance1 - fee)`

- What about this one?
Conventions for Schedules

• We'll 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)
    ```

• when it comes to maintaining isolation, we don’t care about what a program does in its own address space
• we don’t care about the actual values, so we'll just use a different variable for each data item that is read or written in the context of a transaction

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

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(B)</td>
<td></td>
</tr>
<tr>
<td>w(A)</td>
<td>r(A)</td>
<td></td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
<td></td>
</tr>
</tbody>
</table>

• We can also write a schedule on a single line using this notation:
  r₁(A) = transaction T₁ reads A
  w₁(A) = transaction T₁ writes A
• example for the table above:
  r₁(A);  r₂(B);  w₁(A);  r₂(A);  w₂(A)
Conflict Serializability

• Rather than ensuring serializability, it’s easier to ensure a stricter condition known as conflict serializability.

• A conflict in a schedule is a pair of actions (reads or writes) that cannot be swapped without potentially changing the behavior of one or more of the transactions in the schedule.

  • example: in the schedule at right, T1's write of A and T2's read of A conflict. why?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td></td>
<td>w(A)</td>
</tr>
</tbody>
</table>

• A schedule is conflict serializable if we can turn it into an equivalent serial schedule by swapping pairs of consecutive actions that don’t conflict.

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:
  • $w_i(A); r_j(A)$ 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 tx (their order is fixed by the client)

• Pairs of actions that don’t conflict:
  • $r_i(A); r_j(A)$
  • $r_i(A); r_j(B)$
  • $r_i(A); w_j(B)$
  • $w_i(A); r_j(B)$
  • $w_i(A); w_j(B)$
Example of a Conflict Serializable Schedule

\[
\begin{array}{c}
 r_2(A); r_1(A); r_2(B); w_1(A); w_2(B); r_1(B); w_1(B) \\
 r_2(A); r_2(B); r_1(A); w_2(B); r_1(B); w_1(B) \\
 r_2(A); r_2(B); r_1(A); w_1(A); r_1(B); w_1(B) \\
 r_2(A); r_2(B); w_2(B); r_1(A); r_1(B); w_1(B)
\end{array}
\]

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, but not all serializable schedules are conflict serializable

Consider the following schedule involving three txns:

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
<td>w(B)</td>
</tr>
</tbody>
</table>

- It *is* serializable because it is equivalent to either \( T_1; T_2; T_3 \) or \( T_2; T_1; T_3 \) why?

- It’s *not* conflict serializable, because we can’t swap either:
  - \( w_1(A) \) with \( w_2(A) \) – and thus can’t move \( T_1 \) after \( T_2 \)
  - \( w_1(A) \) with \( r_2(A) \) – and thus can’t move \( T_1 \) before \( T_2 \)
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 $w_1(A) \ldots r_2(A)$, T1 must come before T2 in any equivalent serial schedule

• To see if we can create an equivalent serial schedule, determine all such constraints and make sure they aren’t contradictory.

• Example: $r_2(A); r_1(A); r_2(B); w_1(A); w_2(B); r_1(B); w_1(B)$

  - $r_2(A) \ldots w_1(A)$ means T2 must come before T1
  - $r_2(B) \ldots w_1(B)$ means T2 must come before T1
  - $w_2(B) \ldots r_1(B)$ means T2 must come before T1
  - $w_2(B) \ldots w_1(B)$ means T2 must come before T1

  Thus, this schedule is conflict serializable.

Testing for Conflict Serializability (cont.)

• What about this schedule? $r_1(B); w_1(B); r_2(B); r_2(A); w_2(A); r_1(A)$

  - $w_1(B) \ldots r_2(B)$ means T1 must come before T2
  - $w_2(A) \ldots r_1(A)$ means T2 must come before T1

  Thus, this schedule is not conflict serializable.
The algorithm for this test makes use of a precedence graph.
• the vertices are the transactions
• add an edge for each precedence constraint: \( T_1 \rightarrow T_2 \) means \( T_1 \) must come before \( T_2 \) in an equivalent serial schedule

Example: \( \text{r}_2(\text{A}); \text{r}_3(\text{B}); \text{w}_4(\text{A}); \text{w}_2(\text{B}); \text{r}_3(\text{B}) \)

\( \text{r}_2(\text{A}) \ldots \text{w}_4(\text{A}) \) means \( T_2 \rightarrow T_4 \)
\( \text{r}_3(\text{A}) \ldots \text{w}_4(\text{A}) \) means \( T_3 \rightarrow T_4 \)
\( \text{r}_1(\text{B}) \ldots \text{w}_2(\text{B}) \) means \( T_1 \rightarrow T_2 \)
\( \text{w}_2(\text{B}) \ldots \text{r}_3(\text{B}) \) means \( T_2 \rightarrow T_3 \)

Algorithm:
• construct the precedence graph
• test for cycles (i.e., paths of the form \( A \rightarrow \ldots \rightarrow A \))
• if the graph is acyclic (no cycles), the schedule is conflict serializable

When a precedence graph is acyclic, any topological ordering of the vertices gives an equivalent serial schedule.

• topological = an ordering of a graph's vertices such that, if there is directed edge from \( A \) to \( B \) (\( A \rightarrow B \)), \( A \) comes before \( B \) in the ordering

For our previous graph, there is only one such ordering:
\( T_1; T_2; T_3; T_4 \)

To build a topological ordering:
• vertices that are "sinks" (have only incoming arrows) come at the end of the ordering
• work backwards from them
• see algorithms for topological sort for more info.
More Examples

- Determine if the following are conflict serializable:
  - \( r_1(A); r_3(A); w_1(A); w_2(A); r_2(B); w_3(B); w_4(B) \)

  ![Diagram](https://example.com/diagram1)

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

  ![Diagram](https://example.com/diagram2)

Recoverability

- While serializability is important, it isn’t enough!
- Consider the schedule at right, which includes “c” actions that indicate when the transactions commit.
  - the schedule is clearly serializable
  - however, what if the system crashes \textit{after} T1’s commit but \textit{before} T2’s commit?
    - during recovery, the system restores T1’s changes as needed, because T1 committed before the crash
    - the system undoes all changes by T2, because it did not commit before the crash
    - problem?
      - we say that this schedule is \textit{not recoverable}
  - In a recoverable schedule, if T1 reads a value written by T2, then T1 must commit \textit{after} T2 commits.
Dirty Reads and Cascading Rollbacks

- Dirty data is data written by an uncommitted txn.
  - it remains dirty until the txn either:
    - commits: in which case the data is no longer dirty and it is safe for other txns to read it
    - aborts: 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

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(B)</td>
<td>w(A)</td>
<td>...</td>
</tr>
<tr>
<td>w(A)</td>
<td>r(C)</td>
<td>w(C)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Dirty Reads and Cascading Rollbacks (cont.)

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(B)</td>
<td>w(B)</td>
</tr>
<tr>
<td>w(B)</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

- Could the revised schedule lead to a cascading rollback?

- To get a cascadeless schedule, don’t allow dirty reads.
Transactions in SQL

- A client can group a collection of SQL commands into a transaction using the commands we mentioned earlier (BEGIN WORK, COMMIT, ROLLBACK).
  - *user-initiated transaction*

- In addition, the low-level operations that are part of a single SQL command are automatically wrapped in their own txn.
  - *example:*
    
    \[
    \text{UPDATE Foo SET bar = 7 WHERE baz < 3;}
    \]

  - *per-command transaction*

- When there is a user-initiated transaction, the per-command transactions are *nested* inside the user-initiated transaction.
  - if it is aborted, they are as well

Extra Practice

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

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(B)</td>
</tr>
<tr>
<td>c</td>
<td>r(A)</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>
Concurrent Control

Review: Properties of Transactions

- 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 other concurrent transactions
  - Durability: once it completes, its changes survive failures

- To guarantee isolation, a DBMS has to prevent problematic interleavings like this one:

```plaintext
txn 1
  read balance1
  read balance2
  if (balance1+balance2 < min)
    write(balance1 - fee)

txn 2
  read balance1
  read balance2
  if (balance1+balance2 < min)
    write(balance1 - fee)
```

- We'll now look at mechanisms that are used to do so.
Concurrency Control Goals

• The mechanisms used to prevent problematic interleavings are known as concurrency control mechanisms.
  • they control the actions of concurrent transactions

• Goals: ensure that the schedule of actions that results from a set of concurrent txns is:
  • **serializable**: equivalent to some serial schedule
  • **recoverable**: ordered so that the system can safely recover from a crash or undo an aborted transaction
    • need to ensure that a txn does not commit before a txn whose write it has read
  • **cascadeless**: ensure that an abort of one transaction does not produce a series of cascading rollbacks
    • need to prevent dirty reads

A Simple Lock-Based Scheme

• Use locks to control access to database elements.
  • lock pages, records, or possibly even entire tables

• We'll start with a simple scheme:
  • one lock per data element
  • before accessing a data element A, a transaction T1 must first request and acquire the lock for A
    • we say that T1 “locks A”
  • if T2 already holds the lock for A, T1 must wait until T2 releases the lock
    • we say that T2 “unlocks A”
A Simple Lock-Based Scheme (cont.)

- We expand our notation for schedules:
  - \( r_i(A) = \text{transaction } T_i \text{ reads } A \)
  - \( w_i(A) = \text{transaction } T_i \text{ writes } A \)
  - \( c_i = \text{transaction } T_i \text{ commits} \)
  - \( l_i(A) = \text{transaction } T_i \text{ requests a lock for } A \)
  - \( u_i(A) = \text{transaction } T_i \text{ unlocks } A \)

- Example:

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l(B); r(B) )</td>
<td>( l(A); r(A) )</td>
</tr>
<tr>
<td>( l(A); r(A) )</td>
<td>( w(A); u(A) )</td>
</tr>
<tr>
<td>( w(A) )</td>
<td>( w(B) )</td>
</tr>
<tr>
<td>( u(A); u(B) )</td>
<td>( )</td>
</tr>
</tbody>
</table>

- As necessary, the DBMS *denies* lock requests for data elements that are currently locked.
  - make the txn wait until the other txn releases the lock, and then grant the lock
  - we'll show a second request for the lock, even though the txn may not actually need to make one

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l(X) )</td>
<td>( l(X) )</td>
</tr>
<tr>
<td>( r(X) )</td>
<td>( denied; wait for T1 )</td>
</tr>
<tr>
<td>( w(X) )</td>
<td>( )</td>
</tr>
<tr>
<td>( u(X) )</td>
<td>( granted )</td>
</tr>
<tr>
<td>( )</td>
<td>( r(X) )</td>
</tr>
<tr>
<td>( )</td>
<td>( u(X) )</td>
</tr>
</tbody>
</table>
Locking and Serializability

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

\[
\begin{array}{c}
T_1 \\
\text{read balance}_1 \\
\text{write(balance}_1 - 500) \\
\text{read balance}_2 \\
\text{write(balance}_2 + 500)
\end{array}
\]
\[
\begin{array}{c}
T_2 \\
\text{read balance}_1 \\
\text{read balance}_2 \\
\text{if (balance}_1 + \text{balance}_2 < \text{min}) \\
\text{write(balance}_1 - \text{fee})
\end{array}
\]

Two-Phase Locking (2PL)

• To ensure serializability, systems use two-phase locking (2PL).
• 2PL requires that all of a txn’s lock actions come before all its unlock actions.
• Two phases:
  • lock-acquisition phase (aka the "growing" phase): locks are acquired, but no locks are released
  • lock-release phase (aka the "shrinking" phase): once a lock is released, no new locks can be acquired
• Reads and writes can occur in both phases.
Two-Phase Locking (2PL) (cont.)

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(bal1); r(bal1); w(bal1); u(bal1)</td>
<td>l(bal1); r(bal1); w(bal1); u(bal1); u(bal2)</td>
</tr>
<tr>
<td>l(bal2); r(bal2); w(bal2); u(bal2)</td>
<td></td>
</tr>
</tbody>
</table>

2PL would prevent this interleaving. why?

An Informal Argument for 2PL’s Correctness

• 2PL always produces conflict serializable schedules.

• Consider schedules involving only two transactions. To get one that is not conflict serializable, we need:
  1) at least one conflict that requires T1 \(\rightarrow\) 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 \(\rightarrow\) T1
     • T2 operates first on the data item in this conflict
     • T2 must unlock it before T1 can lock it: \(u_2(B) .. l_1(B)\)

• To get both 1 and 2, at least one of the txns would have to acquire a lock after it has already released a lock.
  • example: \(.. u_1(A) .. l_2(A) .. u_2(B) .. l_1(B) ..\)
  • this isn’t possible under 2PL
Deadlock

- Consider the following schedule:

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(B); r(B)</td>
<td>l(A); w(A)</td>
</tr>
<tr>
<td>l(A)</td>
<td>(B) denied; wait for T₁</td>
</tr>
<tr>
<td>denied; wait for T₂</td>
<td></td>
</tr>
</tbody>
</table>

- 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 deadlocks.

Shared vs. Exclusive 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:
  - *shared locks* (also called *read locks*):
    - allow a txn to read a data item
    - multiple txns can hold a shared lock for the same data item at the same time
    - to acquire a shared lock for a data item D, no other txn may hold an exclusive lock for D
  - *exclusive locks* (also called *write locks*):
    - allow a txn to read or write a data item
    - only one txn can hold an exclusive lock for a given data item
    - to acquire an exclusive lock for a data item D, no other txn may hold *any* lock for D
Shared vs. Exclusive Locks (cont.)

• New operations for our schedules:
  sl(A) = transaction Ti requests a shared lock for A
  xl(A) = transaction Ti requests an exclusive lock for A

• Examples:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>xl(A); w(A)</td>
<td></td>
</tr>
<tr>
<td>sl(B); r(B)</td>
<td>sl(B); r(B)</td>
</tr>
<tr>
<td>xl(C); r(C)</td>
<td>u(A); u(B)</td>
</tr>
<tr>
<td>w(C)</td>
<td></td>
</tr>
<tr>
<td>u(B); u(C)</td>
<td></td>
</tr>
</tbody>
</table>

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

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

T1  | T2          | granted or denied?
-----|-------------|------------------------
xl(A); sl(B) | sl(A)       | granted or denied?
sl(B) | granted or denied? |
we(A) | granted or denied? |
xl(B) | granted or denied? |
sl(A) | granted or denied? |

What About Recoverability?

• 2PL alone does not guarantee recoverability, nor does it prevent cascading rollbacks.

• Example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>2PL?</th>
</tr>
</thead>
<tbody>
<tr>
<td>xl(A); r(A)</td>
<td>xl(A); w(A)</td>
<td>recoverable? why or why not?</td>
</tr>
<tr>
<td>w(A); u(A)</td>
<td>xl(C); r(C)</td>
<td></td>
</tr>
<tr>
<td>u(A)</td>
<td>u(A)</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>w(C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>u(C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
Strict Locking

- To ensure that a schedule is recoverable and cascadeless, we need to also employ strict locking.
- make txns hold all exclusive locks until they commit or abort
- doing so prevents dirty reads, which means schedules are always recoverable and cascadeless

\[
\begin{array}{|c|c|}
\hline
T_1 & T_2 \\
\hline
xl(A); r(A) & xl(A); w(A) \\
w(A); u(A) & xl(C); r(C) \\
commit & u(A) \\
\hline
w(C) & xl(C); r(C) \\
\hline
u(C) & u(A) \\
commit & u(C) \\
\hline
\end{array}
\]

- strict + 2PL = strict 2PL

Rigorous Locking

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

\[
\begin{array}{|c|c|c|}
\hline
T_1 & T_2 & T_3 \\
\hline
sl(A); r(A) & xl(A); w(A) & sl(A); r(A) \\
\hline
u(A) & commit & u(A) \\
\hline
\hline
\ldots & xl(A); r(A) & \ldots \\
\hline
\hline
\ldots & commit & print A \\
\hline
\hline
\end{array}
\]

- 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
### Lock Upgrades

- It can be problematic to acquire an exclusive lock earlier than necessary.
  - example: top right

- Instead:
  - acquire a shared lock first
  - only acquire the exclusive lock when you need it.
  - example: bottom right

- Adding an exclusive lock to a shared lock for the same item is known as a *lock upgrade*.
  - need to wait if others hold shared locks for the item

<table>
<thead>
<tr>
<th>Transaction 1 (T1)</th>
<th>Transaction 2 (T2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xl(A) r(A)</td>
<td>sl(A)</td>
</tr>
<tr>
<td>VERY LONG</td>
<td></td>
</tr>
<tr>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>w(A) u(A)</td>
<td></td>
</tr>
</tbody>
</table>

*sl(A) waits a long time for T1!*

<table>
<thead>
<tr>
<th>Transaction 1 (T1)</th>
<th>Transaction 2 (T2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(A) r(A)</td>
<td></td>
</tr>
<tr>
<td>VERY LONG</td>
<td></td>
</tr>
<tr>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>xl(A)</td>
<td></td>
</tr>
<tr>
<td>w(A) u(A)</td>
<td></td>
</tr>
</tbody>
</table>

*r(A) finally!*

### Lock Compatibility Matrices

- Used to determine if a lock request should be granted.

- When there are only shared and exclusive locks, we get:

<table>
<thead>
<tr>
<th>Mode of existing lock (by other txn)</th>
<th>Mode requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>shared, exclusive</td>
</tr>
<tr>
<td>exclusive</td>
<td>shared, exclusive</td>
</tr>
</tbody>
</table>

- Check all rows that apply, since one txn may hold both a shared and exclusive lock for the same item (after an upgrade).
A Possible 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 waits for the other to release its shared lock
  - deadlock!

- Example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(A)</td>
<td>sl(A)</td>
</tr>
<tr>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>xl(A)</td>
<td>xl(A)</td>
</tr>
<tr>
<td>denied; wait for T2</td>
<td>denied; wait for T1</td>
</tr>
</tbody>
</table>

Update Locks

- To avoid deadlocks from lock upgrades, some systems take the following approach:
  - don’t allow the upgrading of shared locks
  - provide a third lock mode known as an update lock
    - like a shared lock, it allows a txn to read an item
    - it can be upgraded to an exclusive lock

- if read-only → acquire a shared lock
- if read-modify-write → acquire an update lock for the read, (RMW) and upgrade it to exclusive for the write
Update Locks (cont.)

- Lock compatibility matrix with update locks:

<table>
<thead>
<tr>
<th>mode of existing lock</th>
<th>shared</th>
<th>exclusive</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>exclusive</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>update</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

- 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 from upgrades
- If a txn holds an update lock on an item, other txns can't acquire other locks on that item.
  - prevents the RMW txn from waiting indefinitely to upgrade

Example of Using Update Locks

| T_1 | T_2 | Example (ul(A) = T, requests an update lock for A):
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(B); ul(C)</td>
<td>ul(A)</td>
<td>granted: the only existing lock on B is a shared lock</td>
</tr>
<tr>
<td></td>
<td>ul(B)</td>
<td>denied: there is already an update lock on C</td>
</tr>
<tr>
<td></td>
<td>u(C)</td>
<td>denied: shared locks cannot be upgraded under this scheme</td>
</tr>
<tr>
<td>u(B)</td>
<td>xl(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xl(B)</td>
<td>granted: update locks can be upgraded if no other locks</td>
</tr>
</tbody>
</table>
Handling Deadlocks: Detection

• One way of handling deadlocks is to have the DBMS detect them and abort one of the transactions involved.

• To detect deadlocks, use a *waits-for graph*.
  • the vertices are the transactions
  • an edge from T1 → T2 means that T1 is waiting for T2 to release a lock
  • a cycle in the graph indicates a deadlock

• Example:

```
T1  T2  T3
xl(A)  sl(B)  xl(C)
denied; wait for T2  sl(C)  denied; wait for T3
```

![Diagram of waits-for graph]

Another Example

• Would the following schedule produce deadlock?

```
T1  T2  T3
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 a lock for an item is acquired just before it is first needed

```
T1  T2  T3
```

![Diagram of another waits-for graph]
Handling Deadlocks: Prevention

- Another set of techniques attempt to prevent deadlocks.
  1. lock ordering: require that txns acquire locks in some order
     - example: the order defined by the keys
     - this prevents deadlocks that don't involve lock upgrades

  2. timeouts: set a maximum time that txns are allowed to wait
     - if a txn waits longer, it times out and is aborted
     - this doesn’t completely prevent deadlocks, it just prevents them from lasting indefinitely!

  3. timestamps: assign a timestamp to a transaction when it begins
     - when T1 needs to wait for T2 to release a lock, decide what to do based on their timestamps
     - may abort T1 or T2 instead of making T1 wait
     - use a policy that prevents deadlocks
     - for more info, look up wound-wait and wait-die

Starvation

- *Starvation* occurs when a transaction never completes.

- Example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(A)</td>
<td>sl(B)</td>
<td></td>
</tr>
<tr>
<td>xl(B)</td>
<td>xl(A)</td>
<td>sl(B)</td>
</tr>
<tr>
<td><em>denied; wait for T2</em></td>
<td><em>denied; wait for T1</em></td>
<td></td>
</tr>
<tr>
<td><strong>deadlock – abort</strong></td>
<td><strong>deadlock</strong></td>
<td></td>
</tr>
</tbody>
</table>

- Most deadlock-handling schemes need an additional policy to prevent this.
**Starvation (cont.)**

- Starvation can also occur in the absence of deadlocks.
  - example:
    - T1 acquires a shared lock for A
    - T2 requests an exclusive lock for A and waits
    - T3 acquires a shared lock for A
    - T1 unlocks A (but T2 still needs to wait for T3!)
    - T4 acquires a shared lock for A
    - T3 unlocks A (but T2 still needs to wait for T4!)
    - ...

- What policy regarding the granting of lock requests would prevent starvation in the absence of deadlocks?

---

**Locking in Tree Index Structures**

- Using 2PL on internal nodes of a tree would be problematic, because there are only a small number of them at the higher levels of the tree.

```
3 10 14 20 28 34 40 51 61 68 77 80 87 90 93 97
```

- In particular, holding an exclusive lock on the root node would prevent any other transaction from accessing the entire tree!

- Instead, systems typically use *lock coupling*.
  - while holding a lock for page P, acquire a lock for P’s child
  - if P’s child is not full, release the lock for P
  - if P’s child is full, we keep the lock for P, since a split might propagate back up to P
Phantom Phenomenon

- Consider the following interleaving:

\[
T1 \quad \begin{array}{l}
\text{for all accts at branch 1} \\
\quad \text{read(acct.bal)} \\
\quad \text{total} += \text{acct.bal}; \text{count} += 1 \\
\quad \text{write(total/count)};
\end{array}
\]

\[
T2 \quad \begin{array}{l}
\text{insert a new acct at branch 1} \\
\text{insert a new acct at branch 2} \\
\text{commit}
\end{array}
\]

\[
\text{for all accts at branch 2}
\]

- It may be allowed by 2PL, even though the result is not equivalent to either serial ordering.
  - if locking records, txns would be allowed; if locking pgs, they might be
  - a similar problem can happen when records are deleted

- The record for the new account at branch 1 is a phantom record.
  - should be seen by T1, but it isn’t

- Conflict serializability does not guarantee serializability in the face of insertions and deletions.

Phantom Phenomenon (cont.)

- How can we prevent interleavings like this one?

\[
T1 \quad \begin{array}{l}
\text{for all accts at branch 1} \\
\quad \text{read(acct.bal)} \\
\quad \text{total} += \text{acct.bal}; \text{count} += 1 \\
\quad \text{write(total/count)};
\end{array}
\]

\[
T2 \quad \begin{array}{l}
\text{insert a new acct at branch 1} \\
\text{insert a new acct at branch 2} \\
\text{commit}
\end{array}
\]

\[
\text{for all accts at branch 2}
\]

- Two cases to consider:
  1) there’s an index on the attribute involved the selection (in this case, the branch attribute)
     → use index locking: lock the index page(s) for the selected value(s) at start of txn
  2) there isn’t an index on that attribute
     → use a special per-table lock for queries that affect/are affected by phantom phenom.
Optimistic Concurrency Control

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

    | T₁  | T₂       |
    |-----|---------|
    | sl(B); r(B) | xl(A); w(A) |
    | ... | xl(B) |

  - Denied, because T₁ might read B again.

- There are other approaches that are *optimistic*.
  - They assume serializability will be maintained.
  - They 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.
  - \( TS(T₂) > TS(T₁) \) if and only if \( T₂ \) started after \( T₁ \).
  - These timestamps are different than the ones used for deadlock prevention.

- The system ensures that all operations are consistent with a serial ordering based on the timestamps.
  - Only allows actions if they are consistent with a schedule in which the transactions execute instantaneously at the time at which they start.
Timestamp-Based Concurrency Control (cont.)

• Examples of actions that are not allowed:
  • example 1:
    
    | T1 | T2 |
    |----|----|
    | TS = 102 | TS = 100 |
    | w(A) | r(C) |
    | r(A) |    |

  not allowed. data item A was written by T1, which should execute after T2 in a serial order based on their timestamps, and thus T2 should not be able to see this value. We say that T2’s read is too late.

  • example 2:
    
    | T3 | T4 |
    |----|----|
    | TS = 209 | TS = 205 |
    | r(B) | r(A) |
    | w(B) |    |

  not allowed. T3 should execute after T4 in a serial ordering. Thus, T3 should have read this value of B, but instead it read the prior value. We say that T4’s write is too late.

Timestamp-Based Concurrency Control (cont.)

• When a txn attempts to perform an action that is inconsistent with a timestamp ordering:
  • the offending txn is aborted
  • 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

  • Aborting the txn ensures that all of its actions will 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:

- read timestamp: RTS(A) = the largest timestamp of any txn that has read A
- write timestamp: WTS(A) = the largest timestamp of any txn that has written A

Timestamp Rules for Reads

If T tries to read A, the system compares TS(T) and WTS(A).

- if TS(T) < WTS(A), abort 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))
**Timestamp Rules for Writes**

- If T tries to write A, the system compares TS(T) with both RTS(A) and WTS(A).
  - if TS(T) < RTS(A), abort 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 example 2)
  - else if TS(T) < WTS(A), ignore the write and let T continue
    - T comes before the txn that wrote A, and thus T’s write should have come first
    - if T’s write had come first, it would have been overwritten, so we can ignore it
  - 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. what iftxn 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 aborted because TS(T) < RTS(A)
    - if U tries to read A after we ignore T’s write:
      - U will be aborted because TS(U) < WTS(A)
Example of Using Timestamps

• Would they prevent our problematic balance-transfer example?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>bal1</th>
<th>bal2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS = 350</td>
<td>TS = 375</td>
<td>RTS = WTS = 0</td>
<td>RTS = WTS = 0</td>
</tr>
<tr>
<td>r(bal1)</td>
<td>r(bal1); r(bal2)</td>
<td>RTS = 350</td>
<td>RTS = 375</td>
</tr>
<tr>
<td>w(bal1)</td>
<td>w(bal1)</td>
<td>WTS = 350</td>
<td>WTS = 375</td>
</tr>
<tr>
<td>r(bal2)</td>
<td>w(bal2)</td>
<td>RTS: no change</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTS = 375</td>
<td></td>
</tr>
</tbody>
</table>

what’s the problem here?

Preventing Dirty Reads Using a Commit Bit

• To prevent dirty reads when using timestamps, we associate a *commit bit* with each data element A.
  • initially, A.commit is true
  • when txn T writes A:
    • set A.commit to false, since T hasn't committed yet
    • update WTS(A) as before
    • A.commit remains false until the most recent writer of A commits or aborts
  • When A.commit == false, we make a txn U wait if:
    • it would be allowed to read A
    • it tries to write A and its write would be ignored
    • its write may be needed if T is rolled back
  • *We don't* make a txn wait if its write of A is allowed.
    • it becomes the most recent writer of A
Preventing Dirty Reads Using a Commit Bit (cont.)

- When a txn T commits:
  - set to true the commit bits of all data elements of which T is the most recent writer
  - allow waiting transactions to proceed

- When atxn 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

---

Example of Using Timestamps and Commit Bits

- The balance-transfer example would now proceed differently.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>bal1</th>
<th>bal2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS = 350</td>
<td>TS = 375</td>
<td>RTS = WTS = 0</td>
<td>RTS = WTS = 0</td>
</tr>
<tr>
<td>r(bal1)</td>
<td>r(bal1)</td>
<td>c = true</td>
<td>c = true</td>
</tr>
<tr>
<td>w(bal1)</td>
<td>r(bal1)</td>
<td>RTS = 350</td>
<td>RTS = 350</td>
</tr>
<tr>
<td>r(bal2)</td>
<td>w(bal1)</td>
<td>WTS = 350; c = false</td>
<td>WTS = 350; c = false</td>
</tr>
<tr>
<td>w(bal2)</td>
<td>r(bal1)</td>
<td>and completes</td>
<td>c = true</td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td>RTS = 375</td>
<td>RTS = 350</td>
</tr>
</tbody>
</table>

CAS CS 460/660, Fall 2013

David G. Sullivan, Ph.D.
### Another Example

- How will this schedule be executed?
  
  \[ r_1(B); r_2(B); w_1(B); w_3(A); w_2(A); w_3(B); \text{commit}_3; r_2(A) \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>RTS = WTS = 0</td>
<td>c = true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RTC = WTS = 0</td>
<td>c = true</td>
</tr>
</tbody>
</table>

### Example 3

- How will this schedule be executed?
  
  \[ w_1(A); w_2(A); r_3(B); w_3(B); r_3(A); r_2(B); w_1(B); r_2(A) \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>RTS = WTS = 0</td>
<td>c = true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RTC = WTS = 0</td>
<td>c = true</td>
</tr>
</tbody>
</table>
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 is given the version of A that it should read, based on the timestamps.
  - the DBMS never needs to roll back a read-only transaction!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>A(0)</th>
<th>A(105)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS = 105</td>
<td>TS = 101</td>
<td>RTS = WTS = 0</td>
<td>c = true; val = &quot;foo&quot;</td>
<td></td>
</tr>
<tr>
<td>r(A)</td>
<td>w(A)</td>
<td></td>
<td>RTS = 105</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>r(A): get A(0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TS = 112</td>
<td>RTS = 0; WTS = 105</td>
<td>c = false; val = &quot;bar&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r(A)</td>
<td>get A(105)</td>
<td>c = true</td>
<td></td>
</tr>
</tbody>
</table>

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)?

<table>
<thead>
<tr>
<th>A(0)</th>
<th>A(105)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS = 75</td>
<td>RTS = 0</td>
</tr>
</tbody>
</table>
Multiversion Timestamp Protocol (cont.)

- Let’s say that a write would be ignored under the standard timestamp protocol. Should it be still be ignored?

- Versions can be discarded as soon as there are no active transactions that could read them.
  - can discard A(t1) if these two conditions hold:
    - there is another, later version, A(t2), with t2 > t1
    - there is no active transaction T with a TS < t2

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

- Some commercial systems use a combination of two-phase locking (2PL) and multiversion timestamping.
- Transactions that perform writes use 2PL.
- Multiple versions of data elements are maintained, with each write creating a new version, as in multiversion timestamping.
- Read-only transactions are allowed to read the appropriate version, as in multiversion timestamping.

Looking Ahead

- Recall the “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 other concurrent transactions
  - Durability: once it completes, its changes survive failures
- We’ve finished our coverage of how the DBMS provides isolation.
- We’ll look next at how the DBMS guarantees atomicity and durability.
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

- This week, we’ll 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 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's read from disk into the buffer cache, and then 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
    - atomicity: *none* of its changes take effect
    - also used when a transaction is aborted

- In order for recovery to work, need to maintain enough state about txns to be able to redo or undo them.

Log-Based Recovery

- The *log* is a file that stores the info. needed for recovery.

- It contains *log records*, each of which contains information about an operation performed by a transaction.
  - update record: summarizes a write operation
  - records for transaction begin, commit, and abort

- We do *not* need to log read operations.
  - don’t affect the state of the database
  - aren’t relevant to determining what changes need to be undone/redone

- Each log record is assigned a unique *log sequence number* (*LSN*).

- 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:

  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.

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>txn: 1; BEGIN</td>
</tr>
<tr>
<td>150</td>
<td>txn: 1; item: D1; old: 3000; new: 2500</td>
</tr>
<tr>
<td>225</td>
<td>txn: 1; item: D2; old: 1000; new: 1500</td>
</tr>
<tr>
<td>350</td>
<td>txn: 2; BEGIN</td>
</tr>
<tr>
<td>400</td>
<td>txn: 2; item: D3; old: 7200; new: 6780</td>
</tr>
<tr>
<td>470</td>
<td>txn: 1; item: D1; old: 2500; new: 2750</td>
</tr>
<tr>
<td>550</td>
<td>txn: 1; COMMIT</td>
</tr>
<tr>
<td>585</td>
<td>txn: 2; item: D2; old: 1500; new: 1300</td>
</tr>
<tr>
<td>675</td>
<td>txn: 2; item: D3; old: 6780; new: 6760</td>
</tr>
</tbody>
</table>

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, skip updates by txns that *are* on the commit list, as well as begin and abort 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.)**

- The details matter!
  - scanning backward gives the info needed for undo decisions
    - if we encounter an update by a txn that is not on the commit list, we know it must *never* have committed, and thus the update can be undone
    - if we started with a forward scan, we'd need to wait until the end of the scan before any redos could be done
  - to ensure the correct values are on disk after recovery, we:
    - put all redos after all undos
    - perform the undos during a backward pass (reversing the order in which the updates occurred)
    - perform the redos during a forward pass (in the same order in which the updates occurred)
Recovery Using Undo-Redo Logging (cont.)

- Here’s how it would work on our earlier example:

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
<th>backward pass</th>
<th>forward pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>txn: 1; BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>txn: 1; item: D1; old: 3000; new: 2500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>txn: 1; item: D2; old: 1000; new: 1500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td>txn: 2; BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>txn: 2; item: D3; old: 7200; new: 6780</td>
<td></td>
<td></td>
</tr>
<tr>
<td>470</td>
<td>txn: 1; item: D1; old: 2500; new: 2750</td>
<td></td>
<td></td>
</tr>
<tr>
<td>550</td>
<td>txn: 1; COMMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>585</td>
<td>txn: 2; item: D2; old: 1500; new: 1300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>675</td>
<td>txn: 2; item: D3; old: 6780; new: 6760</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recovery restores the database to a consistent state that reflects:
- all of the updates by txn 1
- none of the updates by txn 2
Recovery Using Undo-Redo Logging (cont.)

Here’s how it would work on our earlier example:

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
<th>backward pass</th>
<th>forward pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>txn: 1; BEGIN</td>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>150</td>
<td>txn: 1; item: D1; old: 3000; new: 2500</td>
<td>skip</td>
<td>redo: D1 = 2500</td>
</tr>
<tr>
<td>225</td>
<td>txn: 1; item: D2; old: 1000; new: 1500</td>
<td>skip</td>
<td>redo: D2 = 1500</td>
</tr>
<tr>
<td>350</td>
<td>txn: 2; BEGIN</td>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>400</td>
<td>txn: 2; item: D3; old: 7200; new: 6780</td>
<td>undo: D3 = 7200</td>
<td>skip</td>
</tr>
<tr>
<td>470</td>
<td>txn: 1; item: D1; old: 2500; new: 2750</td>
<td>skip</td>
<td>redo: D1 = 2750</td>
</tr>
<tr>
<td>550</td>
<td>txn: 1; COMMIT</td>
<td>add to commit list</td>
<td>skip</td>
</tr>
<tr>
<td>585</td>
<td>txn: 2; item: D2; old: 1500; new: 1300</td>
<td>undo: D2 = 1500</td>
<td>skip</td>
</tr>
<tr>
<td>675</td>
<td>txn: 2; item: D3; old: 6780; new: 6760</td>
<td>undo: D3 = 6780</td>
<td>skip</td>
</tr>
</tbody>
</table>

Recovery restores the database to a consistent state that reflects:
- all of the updates by tx 1
- none of the updates by tx 2

Logical Logging

We’ve been assuming that update records store old and new data values.
- the associated undo/reduce operations are idempotent
  (can be performed multiple times without changing the result)

It’s also possible to use logical logging, which stores a logical description of the update operation.
- example: increment D1 by 1
- example: insert the substring “foo” at position 100 in a string

Problem: logical update operations are not 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 not been performed, we don’t want to undo it
Logical Logging and LSNs

• To determine if an undo or redo operation should be performed, the DBMS makes use of the log sequence numbers (LSNs).

• When a data element is updated:
  • store the LSN of the update log record with the data element
  • store the old LSN of the data element in the log record

• When recovering, there are three LSNs to consider:
  1) the LSN of the update log record (the record LSN)
  2) the old LSN in the update log record (the olsn)
  3) the LSN stored with the data element (the datum LSN)

Recovery Using LSNs

• Updated rules for recovery under undo-redo logging:
  • backward pass:
    • for each update by a txn not on the commit list, undo the update if the datum LSN == the record LSN
    • when we undo, set the datum LSN = ?
  • forward pass:
    • for each update by a txn on the commit list, redo the update if the datum LSN == the olsn
    • when we redo, set the datum LSN = ?
Recovery Using LSNs

- Updated rules for recovery under undo-redo logging:
  - **backward pass:**
    - for each update by a txn *not* on the commit list, undo the update if the datum LSN == the record LSN
    - when we undo, set the datum LSN = olsn
  - **forward pass:**
    - for each update by a txn on the commit list, redo the update if the datum LSN == the olsn
    - when we redo, set the datum LSN = record LSN

---

Recovery Using LSNs (cont.)

- Example: assume the following datum LSNs:
  - D1: 150
  - D2: 0
  - D3: 400

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
<th>backward pass</th>
<th>forward pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>txn: 1; BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>txn: 1; item: D1; old: 3000; new: 2500; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>txn: 1; item: D2; old: 1000; new: 1500; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td>txn: 2; BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>txn: 2; item: D3; old: 7200; new: 6780; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>470</td>
<td>txn: 1; item: D1; old: 2500; new: 2750; olsn: 150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>550</td>
<td>txn: 1; COMMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>585</td>
<td>txn: 2; item: D2; old: 1500; new: 1300; olsn: 225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>675</td>
<td>txn: 2; item: D3; old: 6780; new: 6760; olsn: 400</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Recovery Using LSNs (cont.)

- Example: assume the following datum LSNs:
  - D1: 150,470
  - D2: 0,225
  - D3: 400,0

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
<th>backward pass</th>
<th>forward pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>tx: 1; BEGIN</td>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>150</td>
<td>tx: 1; item: D1; old: 3000; new: 2500; olsn: 0</td>
<td>skip</td>
<td>150 != 0: don't redo</td>
</tr>
<tr>
<td>225</td>
<td>tx: 1; item: D2; old: 1000; new: 1500; olsn: 0</td>
<td>skip</td>
<td>0 == 0: redo D2 = 1500 datum LSN = 225</td>
</tr>
<tr>
<td>350</td>
<td>tx: 2; BEGIN</td>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>400</td>
<td>tx: 2; item: D3; old: 7200; new: 6780; olsn: 0</td>
<td>400 == 400 undo: D3 = 7200 datum LSN = 0</td>
<td>skip</td>
</tr>
<tr>
<td>470</td>
<td>tx: 1; item: D1; old: 2500; new: 2750; olsn: 150</td>
<td>skip</td>
<td>150 == 150 redo: D1 = 2750 datum LSN = 470</td>
</tr>
<tr>
<td>550</td>
<td>tx: 1; COMMIT</td>
<td>add to commit list</td>
<td>skip</td>
</tr>
<tr>
<td>585</td>
<td>tx: 2; item: D2; old: 1500; new: 1300; olsn: 225</td>
<td>0 != 585 don’t undo</td>
<td>skip</td>
</tr>
<tr>
<td>675</td>
<td>tx: 2; item: D3; old: 6780; new: 6760; olsn: 400</td>
<td>400 != 675 don’t undo</td>
<td>skip</td>
</tr>
</tbody>
</table>

**Recovery Using LSNs: Another Example**

- datum LSNs: D4: 0, D5: 0, D6: 1100, D7: 930

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
<th>backward pass</th>
<th>forward pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>tx: 3; BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>770</td>
<td>tx: 3; item: D5; old: &quot;foo&quot;; new: &quot;bar&quot;; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>825</td>
<td>tx: 4; BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>850</td>
<td>tx: 4; item: D4; old: 9000; new: 8500; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>tx: 4; item: D6; old: 5.7; new: 8.9; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>930</td>
<td>tx: 3; item: D7; old: &quot;zoo&quot;; new: &quot;cat&quot;; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>980</td>
<td>tx: 4; COMMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>tx: 3; item: D4; old: 8500; new: 7300; olsn: 850</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>tx: 3; item: D6; old: 8.9; new: 4.1; olsn: 900</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

---

Checkpoints (cont.)

- **checkpoint interval** = the amount of time between checkpoints
- Recovery time increases as the checkpoint interval increases.
- However, there’s also a cost to performing a checkpoint, so we don’t want to perform them too frequently.
- Typically, systems allow you to tune the checkpoint interval.
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

- 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 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 in the commit list, we can stop the backward pass at the checkpoint record
### Example of Recovery with Dynamic Checkpoints

- **Initial datum LSNs:**
  - D4: 110
  - D5: 0
  - D6: 80

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
<th>backward pass</th>
<th>forward pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>txn: 1; BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>txn: 1; item: D4; old: 20; new: 15; olsn: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>txn: 2; BEGIN</td>
<td>stop here</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>txn: 1; COMMIT</td>
<td>add to commit list</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>txn: 2; item: D5; old: 12; new: 13; olsn: 0</td>
<td>undo: D5 = 12 datum LSN = 0</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>CHECKPOINT (active txns = 2)</td>
<td>note active txns</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>txn: 2; item: D4; old: 15; new: 50; olsn: 110</td>
<td>don't undo</td>
<td>start here</td>
</tr>
<tr>
<td>170</td>
<td>txn: 3; BEGIN</td>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>180</td>
<td>txn: 3; item: D6; old: 6; new: 8; olsn: 80</td>
<td>don't undo</td>
<td>skip</td>
</tr>
</tbody>
</table>

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 changed database pages 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 all dirty database pages 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 dirty database pages must be held in memory until the txn commits and the commit record is forced to disk.

- At transaction commit:
  1. write the commit log record
  2. force all dirty log records to disk
     (dirty 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

- abort records (written after an aborted txn is undone):
  • not crucial: already undo changes by txns w/o a commit record
  • save work during backward pass: if encounter an abort record, don’t need to determine whether to undo thattxn’s changes

- checkpoint records: limit the amount of the log that is processed during recovery
Review: Concurrency Control

- Goals: a schedule of actions by concurrent txns should be:
  - *serializable*
  - *recoverable*
    - ensure onetxn doesn't commit before another txn whoowrite it has read
    - preventing dirty reads would also guarantee this
  - *cascadeless:*
    - prevent dirty reads

- How do the mechanisms that we studied prevent dirty reads?
  - locking:
    - timestamps:

Logging and Concurrency Control

- To implement either of the concurrency-control mechanisms, we need to answer this question:
  When is a txn *really* committed?

- Strictly speaking, a transaction T is not committed until its commit log record makes it to disk.
  - why?

- However, for performance reasons, it can be undesirable to force log records to disk ("flush the log") every time a commit command is received.
Group Commit

- In *group commit*, the DBMS doesn't immediately flush the log every time that a commit command is received.

- Instead, the DBMS:
  - writes the commit record to the in-memory log buffer
  - releases the locks held by the txn or sets the appropriate commit bits to true
  - waits to flush the log until some later point in time
    - the txn's thread/process is put to sleep until then
    - in the meantime, other txns may also commit

- Multiple txns may end up having their commit records written to disk at the same time.
  - hence the name "*group commit*"

Impact of Group Commit

- In *group commit*, it's still the case that a commit operation is not fully complete until the commit record makes it to disk.

- However, we do release locks/reset commit bits before the commit record makes it to disk.

- As a result, there can be dirty reads while a yet-to-be committed txn is waiting for the log to be flushed.

- Can group commit violate atomicity or durability?
  - no, because a txn's log records are still on disk before its commit operation completes
**Impact of Group Commit (cont.)**

- Can group commit violate recoverability?
  - no, provided that the log records are forced to disk in the order in which they were written
- example: let's say that we have the in-memory log buffer shown at right

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>txn: 1; item: D1; old: 30; new: 25</td>
</tr>
<tr>
<td>150</td>
<td>txn: 1; COMMIT</td>
</tr>
<tr>
<td>225</td>
<td>...</td>
</tr>
<tr>
<td>350</td>
<td>txn: 2; COMMIT</td>
</tr>
</tbody>
</table>

- assume T2 reads D1 after T1 writes it
- the database will be in an unrecoverable state if the system crashes with only T2's commit record on disk
  - why?
    - but if the log records are forced to disk in order, this can never happen!
    - if T2's commit record is on disk, T1's must be, too

**Impact of Group Commit (cont.)**

- Can group commit lead to a cascading rollback?
  - no.
  - if a dirty read occurs, the writer of the dirty data has already chosen to commit and will not subsequently choose to abort
  - if a crash occurs, the reader of the dirty data would already have been rolled back as part of recovery – whether or not it performed the dirty read
Extra Practice

- What type of logging is being used to create the log at right?

- What are the possible on-disk values of item D1 given that logging scheme and this log?

- What are the possible on-disk values of D2?

<table>
<thead>
<tr>
<th>LSN</th>
<th>record contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>txn: 1; BEGIN</td>
</tr>
<tr>
<td>150</td>
<td>txn: 1; item: D1; new: 2500</td>
</tr>
<tr>
<td>350</td>
<td>txn: 2; BEGIN</td>
</tr>
<tr>
<td>400</td>
<td>txn: 2; item: D2; new: 6780</td>
</tr>
<tr>
<td>470</td>
<td>txn: 1; item: D1; new: 2750</td>
</tr>
<tr>
<td>550</td>
<td>txn: 1; COMMIT</td>
</tr>
<tr>
<td>585</td>
<td>txn: 2; item: D1; new: 1300</td>
</tr>
</tbody>
</table>

original values:
D1=1000, D2=3000
Course Checkpoint

- Recall our earlier discussion of the two layers of a DBMS:
  - the *logical layer*
  - the *storage layer* or *storage engine*

- We’ve spent the last few weeks looking at how a *transactional* storage engine is implemented.

- Earlier in the course, we studied two logical models:
  - entity-relationship
  - relational

- We’ll now begin looking at other logical models.
Objects in DBMSs: A Bit of History

- In the 1990s, object-oriented languages grew in popularity.

- This led people to attempt to add key aspects of these languages to DBMSs.

- One motivation was the desire to support complex data types:
  - multivalued attributes
  - attributes with structure
    - example: an address
    - want to keep its components separate
    - also want to be able to treat it as a unit
  - types that inherit from other types
  - abstract data types (ADTs) with their own operators/methods
    - including types for nontraditional data like sounds, images, etc.

Objects in DBMSs: A Bit of History (cont.)

- One way: add persistence to an object-oriented language
  - allow objects to persist on disk
  - extend the OO language to support queries, etc.
  - use the language to create an object-oriented DBMS
    - versions of this still exist (e.g., JDO: Java Database Objects)
  - At the time, many people thought that object-oriented DBMSs would replace RDBMSs, but they didn’t.
  - Instead, object-oriented features have been added to relational systems to create object-relational DBMSs.
Object-Oriented Data Models

• We'll look briefly at two different models:
  1. ODL (object definition language)
  2. the object-relational model

• In ODL, relations are not central to the model, although we can express a relation as a set of objects.

• In the object-relational model, relations are still central.

ODL (Object Definition Language)

• a "pure" object-oriented data model

• It was originally intended as the data-definition portion of a query language for OO DBMSs.
  • like CREATE TABLE statements in SQL

• More often, it is used when designing a database.
  • like an ER model
  • even if the resulting database is not object-oriented

• It will allow us to discuss some of the OO features that can be incorporated into DBMSs.
ODL Classes

- Use a separate class for each type of entity.
- Classes have three types of properties:
  - **attributes**: describe an entity
    - can have structure (unlike relational attributes)
  - **relationships**: connections between an object of a class and one or more other objects
  - **methods** that can be applied to objects of the class

```java
class Course {
    attribute string name;
    attribute string semester;
    relationship Room location;
    relationship Set<Student> enrolledStudents;
    Set<string> studentNames();  // method declaration
}
```

- ODL also allows for inheritance: classes can have subclasses.

ODL Data Types

- Basic types:
  - atomic types:
    - integer, float, character, string, boolean, etc.
  - class types
- Structured types, which combine basic types:
  - collection types:
    - `Array<T>`, `Bag<T>`, `List<T>`, `Set<T>`, ...
  - structures:
    - `Struct name { type1 fieldname1, type2 fieldname2, ... }`
      - example:
        - `Struct address { string street, string city, ... }`
Attributes vs. Relationships

- An attribute describes an object.
  - the value of the attribute “belongs to” the object
  - the value cannot be accessed except by means of the object

- A relationship connects an object to one or more independent objects.
  - can be thought of as a set of one or more references to objects that are stored elsewhere

- We don’t typically use class types for attributes.
  - implies that we’re embedding one object inside another one
  - instead, use struct types for attributes with structure
  - an attribute with a class type should really be a relationship!

Capturing Relationships in ODL

- Recall: a relationship set is often translated into a relation.
- example: we would need three relations for the following

  \[
  \text{Course}(\text{id}, \text{name}, \text{numCredits}, \ldots);
  \text{Student}(\text{id}, \text{name}, \text{address}, \ldots);
  \text{Enrolled}(\text{student_id}, \text{course_id});
  \]

- In ODL, we only need two classes: Course and Student.
- capture Enrolled within the class for one or both entity sets:

  ```
  \text{class Course} \{
    \ldots
    \text{relationship Set<Student> enrolledStudents;}
  \}
  ```
Capturing Relationships in ODL (cont.)

- Recall: a relationship set is sometimes captured within the relation for one of the connected entity sets:

```
Course(id, name, ..., room);
Room(id, building, roomNumber, ...)
```

- this works for what types of relationships?

- In ODL, we can also capture many-to-many relationships in the classes of the connected entity sets.
  - example: the `enrolledStudents` relationship

- However, we need a separate "relationship" class for ternary and higher-degree relationships.

Data Types of ODL Relationships

- Two options, depending on the cardinality constraints:
  - a class type:
    - example: a course meets in at most one room
    ```java
class Course {
    ...
    relationship Room location;
    ...
}
```

- a collection type applied to a class type:
  - example: a course can have more than one student
    ```java
class Course {
    ...
    relationship Set<Student> enrolledStudents;
    ...
}
```
Inverse Relationships

• In ODL, we often capture a binary relationship set by putting a relationship property in both of the entity sets’ classes:

```
class Course {
    relationship Set<Student> enrolledStudents;
}
class Student {
    relationship Set<Course> courses;
}
```

• Two relationships that represent the same relationship set are known as inverse relationships.

Inverse Relationships (cont.)

• If relationship R connects an object A with objects B₁, …, Bₙ, then the inverse relationship of R connects each of the objects B₁, …, Bₙ with object A.
  • ex: for a Student s and a Course c, if s is in c.enrolledStudents, then c should be in s.courses

  • Indicate this in ODL as follows:

```
class Course {
    relationship Set<Student> enrolledStudents
        inverse Student::courses;
}
class Student {
    relationship Set<Course> courses
        inverse Course::enrolledStudents;
}
```
Object Identifiers (OIDs) and Primary Keys

- In an OO DBMS, every object is given a unique identifier (OID).
  - allows the DBMS to implement relationships
  - the actual OID values are not visible to users and cannot be queried

- Because of OIDs, primary keys are optional in ODL.

- However, they can still be specified:

```java
class Course (key (name, semester)) {
    attribute string name;
    attribute string semester;
    attribute int numCredits;
    attribute int cost;
    ...
}
```

Object-Relational Model

- Extends the relational model by adding OO features.

- Tuples are treated like objects.

- The added features include:
  - structured types for attributes and relations, including:
    - user-defined types
    - type inheritance
  - collection types – including sets
  - tuple identifiers: like object identifiers in ODL
  - reference types
  - methods
Nested Relations

- Because the object-relational model includes collection types, we can have an attribute that is a set – i.e., a relation.
- thus, we end up with nested relations
- example:

```plaintext
Courses(name, semester, numCredits, location, enrolledStudents(name, yearOfGrad, ...));
```

```
<table>
<thead>
<tr>
<th>name</th>
<th>semester</th>
<th>...</th>
<th>enrolledStudents</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscie119</td>
<td>fall</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Turing, Alan 2010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sullivan, Perry 2022</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>cscie268</td>
<td>spring</td>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Codd, Ted 2008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sullivan, Perry 2022</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```

Reference Types

- Our schema for `Courses` on the previous page is not ideal.
  - why?

- To address these problems, the object-relational model allows an attribute to store a reference (or a collection of references) to tuples from other relations.
Reference Types (cont.)

• Here’s an improved schema for Courses that uses reference types:

  Courses(name, semester, numCredits,
             location(*Room), enrolledStudents({*Student}));

  • the * indicates a reference
  
  • location is a reference to a single Room tuple

  • enrolledStudents is a set of references to Student tuples (as indicated by the braces)

• Attributes with reference types are like relationships in ODL.

The Object-Relational Model and SQL

• Support for many – but not all – of the features of the object-relational model have been added to SQL.

• The SQL-99 standard includes some object-relational features.
  • a notable exception is that it does not support nested relations

• The SQL-2003 standard adds support for nested relations.
User-Defined Types (UDTs)

- User-defined types are similar to classes in ODL.
  - can include attributes and methods
  - however, they don't have separate relationship properties
  - in addition, they can't include primary keys
    - instead, the keys are part of the definition of the relation

- Syntax: `CREATE TYPE typename AS (attributes) methods;`

- Example:
  `CREATE TYPE AddressType AS (street VARCHAR(30), city VARCHAR(20), zip CHAR(5)) METHOD getState() RETURNS CHAR(2);`

- As in ODL, methods are declared in the type definition, but they are defined separately.
- we won't cover method definitions in this course

User-Defined Types (UDTs) (cont.)

- We can use a UDT as the type of:
  - an attribute:
    `CREATE TYPE StudentType AS (name VARCHAR(30), address AddressType, ...);
    • we say that such attributes have a structured type or row type
  - a relation:
    `CREATE TYPE CourseType AS (name VARCHAR(30), semester CHAR(6), ...
    );
    CREATE TABLE Courses OF CourseType;`
User-Defined Types (UDTs) (cont.)

- If a UDT is used as the type of a relation, the primary key is specified in the `CREATE TABLE` statement as follows:

```sql
CREATE TABLE Courses OF CourseType (  
    PRIMARY KEY(name, semester)  
);  
```

References

- We can define an attribute to be a reference to a tuple from a relation of a given type.

```sql
CREATE TYPE RoomType AS (  
    building VARCHAR(30), roomNumber CHAR(6));
CREATE TYPE CourseType AS (  
    name VARCHAR(30), semester CHAR(6), ...  
    location REF(RoomType), ...  
);  
```

- We can also specify that the referenced tuple must come from a particular relation of that type:

```sql
CREATE TABLE Classrooms OF RoomType;
CREATE TABLE Courses OF CourseType (  
    location REF(RoomType) SCOPE Classrooms);  
```
Reference Columns

- For tuple identifiers, SQL uses values from a special reference column.

- The values stored in this column can be:
  - system-generated
  - user-generated
  - derived from the primary key

- We'll limit ourselves to system-generated reference columns, which are specified when creating the table:

  ```sql
  CREATE TABLE Students OF StudentType (REF IS studentID SYSTEM GENERATED);
  ```

- Unlike OIDs in ODL, tuple identifiers are visible to users – just like any other attribute.

Accessing Fields in Structured and Reference Types

- To access a field in an attribute with a structured type, we use the . operator:

  ```sql
  CREATE TYPE StudentType AS (
    name VARCHAR(30),
    address AddressType,
    ...);
  CREATE TABLE Students OF StudentType;
  SELECT S.name, S.address.city
  FROM Students AS S;
  ```

- To access an attribute in a referenced tuple, we use the -> operator:

  ```sql
  CREATE TYPE CourseType AS (...
    location REF(RoomType),...);
  CREATE TABLE Courses OF CourseType ...;
  SELECT C.name, C.location->roomNumber
  FROM Courses AS C;
  ```
References and Queries

- Find the room numbers of all courses that meet in the Sever Hall:
  
  ```sql
  SELECT C.location->roomNumber
  FROM Courses AS C
  WHERE C.location->building LIKE "Sever%";
  ```

- Note that we’re accessing information from tuples in the Classrooms relation without needing to join Courses and Classrooms.

Array Types

- We can specify an attribute that is a fixed-sized array:
  
  ```sql
  CREATE TYPE StudentType AS (
    name VARCHAR(30),
    address AddressType,
    phones CHAR(10) ARRAY[5]
  );
  ```

- There are special operators to:
  - access an array element:
    ```sql
    phones[2]
    ```
  - get the number of elements in the array:
    ```sql
    CARDINALITY(phones)
    ```
Multiset Types

- A *multiset* is an unordered collection in which a given element can appear more than once.
  - also known as a *bag*

- Support for multiset types was added in SQL-2003. Example:
  
  ```sql
  CREATE TYPE StudentType AS (
    name VARCHAR(30),
    address AddressType,
    phones CHAR(10) ARRAY[5],
    courses REF(CourseType) MULTISET
  );
  ```

- Differences between array types and multiset types:
  - elements of an array have an associated position; elements of a multiset do not
  - an array has a fixed size; a multiset does not

Nested Relations in SQL-99

- Nested relations are not really part of SQL-99, because that standard doesn't support sets.

- Instead, we still need to use a separate relation to capture many-to-many relationship sets.

- Example:
  
  ```sql
  CREATE TABLE Students OF StudentType (REF IS studentID SYSTEM GENERATED);
  CREATE TABLE Courses OF CourseType (REF IS courseID SYSTEM GENERATED);
  CREATE TABLE EnrolledIn (student REF(StudentType) SCOPE Students,
                            course REF(CourseType) SCOPE Courses);
  ```
Nested Relations in SQL-2003

• An attribute with a multiset type is a nested relation.

• Thus, we no longer need a separate relation to capture many-to-many relationship sets.

```
CREATE TYPE StudentType AS (  
    name VARCHAR(30),
    address AddressType,
    phones CHAR(10) ARRAY[5],
    courses REF(CourseType) MULTISET
);

CREATE TYPE CourseType AS (  
    name VARCHAR(30),
    semester CHAR(6),
    location REF(RoomType), ...
    students REF(StudentType) MULTISET
);
```

ER Diagram → Object-Relational Schema

• Given the extensions to SQL, we can now capture additional aspects of ER models.

• Example from earlier in the semester:

```
course

exam dates

start time

end time

length

= end time – start time
```

• how would we capture exam dates?

• how would we capture length?
Specifying Complex-Type Values

- When inserting tuples in a relation, how do we specify array, multiset, and structured values?

- We can specify an array or multiset as follows:
  
  ```java
  ARRAY["123-456-7890", "777-666-5555", ...]
  MULTISET["45678", "12345", "67890", ...]
  ```

- We can specify an attribute that has structure as follows:
  ```java
  new AddressType("33 Oxford Street", "Cambridge", "MA", "02138");
  ```

  [assuming the following definition of this type:
  ```java
  CREATE TYPE AddressType AS {
  street VARCHAR(30),
  city VARCHAR(20),
  state CHAR(2),
  zip CHAR(5)};
  ```

Example of Specifying Complex-Type Values

- Recall our `StudentType` definition:
  ```java
  CREATE TYPE StudentType AS {
  name VARCHAR(30),
  address AddressType,
  phones CHAR(10) ARRAY[5],
  courses REF(CourseType) MULTISET
  };
  ```

- If we have a table `Students` of type `StudentType`, how would we insert a tuple in this relation?
  - here's everything but the multiset:
    ```java
    INSERT INTO Students VALUES {
    "Ted Codd",
    new AddressType("IBM Almaden", "San Jose", "CA", "95120"),
    ARRAY["123-456-7890", "444-555-7777"],
    ...}
    ```
Example of Specifying Complex-Type Values (cont.)

• It's tricky to insert the multiset in this case because the values in the reference column are system-generated:

```sql
CREATE TABLE Courses OF CourseType (
    REF IS courseID SYSTEM GENERATED
);
```

• Solution: use a nested query:

```sql
INSERT INTO Students VALUES (
    "Ted Codd",
    new AddressType("IBM Almaden", "San Jose", "CA", "95120"),
    ARRAY["123-456-7890", "444-555-7777"],
    MULTISET(SELECT courseID FROM Course
        WHERE name = "CS165" OR name = "CS182"));
```

Queries Involving Complex-Type Values

• The SQL standards specify a number of special operators that make it easier to work with complex-type values in queries.

• For example:

```sql
SELECT S.name
FROM Student AS S
WHERE '123-456-7890' MEMBER OF S.phones;
```

where S.phones is a multiset.

• Another example:

```sql
SELECT S.name, P.phone
FROM Student AS S , unnest(S.phones) AS P(phone);
```

would create tuples of the form (student-name, student-phone), with a separate tuple for each of the student's phone numbers.
Implementation Issues

• In pure OODBMSs, transferring objects to and from disk involves translating references/pointers:
  memory addresses $\leftrightarrow$ addresses stored on disk
  This process is known as pointer swizzling.

• Other issues arise in both OO and object-relational DBMSs, such as:
  • attributes that have collection types can have arbitrarily large sizes. how do you store them efficiently?
  • what if the methods associated with user-defined types have bugs?

Implementing Array and Multiset Types

• Option 1: use our standard approach for variable-length records, and store the array or multiset within the record.
  
  0 4 8 12 16 23 31
  16 23 31 75 1234567 foobar xx 123 456 789 ...

  an array or multiset

  • can just store an offset to the first element, and use simple arithmetic to access the other elements

• Option 2: store the array or multiset on a separate page, and store a reference to it in the record for the tuple

  0 4 8 12 16 23 31
  16 23 31 39 1234567 foobar xx (25, 2)
Implementing Array and Multiset Types (cont.)

- Option 3: use separate tables for array or multiset attributes.
- example:

<table>
<thead>
<tr>
<th>student_id</th>
<th>name</th>
<th>...</th>
<th>phones</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td>...</td>
<td>123-456-7890, 444-777-0000</td>
</tr>
<tr>
<td>77777777</td>
<td>Ted Codd</td>
<td>...</td>
<td>333-888-2222, 111-666-4444</td>
</tr>
<tr>
<td>12121212</td>
<td>Alan Turing</td>
<td>...</td>
<td>911-101-1100, 411-101-2222</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_id</th>
<th>name</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>Jill Jones</td>
<td></td>
</tr>
<tr>
<td>77777777</td>
<td>Ted Codd</td>
<td></td>
</tr>
<tr>
<td>12121212</td>
<td>Alan Turing</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student_id</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>123-456-7890</td>
</tr>
<tr>
<td>77777777</td>
<td>444-777-0000</td>
</tr>
<tr>
<td>77777777</td>
<td>333-888-2222</td>
</tr>
</tbody>
</table>

...
Structured Data

- So far, we've seen several different data models:
  - the entity-relationship (ER) model
  - the relational model
  - ODL
    - the object-relational model
- All of these models use some type of schema to define the structure of data.
  - ER diagrams
  - relational or object-relational schemas
  - ODL class declarations
- 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 attributes
Semistructured Data

• In semistructured data:
  • there may or may not be a separate schema
  • the schema is not rigid
    • example: addresses
      • some records may have 3 fields: city, state, and zip
      • other records may group all this info. into a single field
  • Semistructured data is self-documenting.
    • information describing the data is embedded with the data
      <course>
        <name>CSCI E-268</name>
        <begin>19:35</begin>
        ...
      </course>

Semistructured Data (cont.)

• Its characteristics 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
In the semistructured-data model, a database is a graph.

- There is a single root node that represents the entire database.
  - There is at least one path from it to every other node
  - May be more than one path, so the graph may not be a tree

- A leaf node represents a value with an atomic type (int, real, etc.).

- A non-root interior node represents an entity, object, or struct.
  - Example: the shaded node represents the course CSCI E-119.

The directed edges in the model:

- Connect the root node to all top-level entities
- Connect an entity to attributes of that entity
  - Ex: CSCI E-119 is connected to its name and start/end times
- Specify relationships between two entities.
  - Ex: CSCI E-119 is connected to its room

- The label on an edge specifies the nature of the connection.
As mentioned earlier, the structure used for a given type of information can vary within a single database.

- CSCI E-268's begin and end times are grouped together into a time struct that is an attribute of the course
- CSCI E-119's start and end times are specified as two separate attributes
- This type of heterogeneity can occur when records from two or more databases are merged into a single database.

**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:
    ```html
    <li>Like HTML, XML is a markup language.</li>
    ``
    - 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>CSCI E-268</name>

• An element can include other nested child elements.
  <course>
    <name>CSCI E-268</name>
    <begin>19:35</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 include attributes that describe it.

• Specified within the element’s begin tag.
  • syntax: name="value"

• Example:
  <course catalog_number="12345" exam_group="16"/>
Attributes vs. Child Elements

<table>
<thead>
<tr>
<th></th>
<th>attribute</th>
<th>child element</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>number of occurrences</strong></td>
<td>at most once in a given element</td>
<td>an arbitrary number of times</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>always a string</td>
<td>can have its own children</td>
</tr>
</tbody>
</table>

- 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
    - the root element may optionally be preceded by an XML declaration that looks something like this:
      ```xml
      <?xml version="1.0" standalone="yes"?>
      ```
    - each child element is completely nested within its parent element
      - this would **not** be allowed:
        ```xml
        <course>
          <name>CSCI E-268</name>
          <time>
            <begin>19:35</begin>
            <end>21:35</end>
          </time>
        </course>
        ```
  - The elements need not correspond to any predefined standard.
    - a separate schema is not required
Example of Well-Formed XML Document

```xml
<?xml version="1.0" standalone="yes"?>
<university-data>
  <course>
    <name>CSCI E-119</name>
    <start>19:35</start>
    <end>21:35</end>
  </course>
  <room>
    <bldg>Sever</bldg>
    <num>203</num>
  </room>
  <course>
    <name>CSCI E-268</name>
    <time>
      <begin>19:35</begin>
      <end>21:35</end>
    </time>
  </course>
  ...
</university-data>
```

XML and Semistructured Data

- XML captures the same structure as semistructured data graphs.

```
<?xml version="1.0" standalone="yes"?>
<university-data>
  <course><name>CSCI E-119</name>
    <start>19:35</start>
    <end>21:35</end>
  </course>
  ...
</university-data>
```

- An element corresponds to a node in the graph.
- A child element corresponds to the child of a node.
- The root element is like the root node in the graph.
- Tags play the same role as edge labels.
XML and Semistructured Data (cont.)

• Nesting elements allows us to capture a semistructured graph that is a tree.
  • at most one path from the root node to a given element
• We’ll see later how to capture non-tree graphs.

XML Namespaces

• Because XML is used to combine info. from different sources, a given document may contain different types of elements with the same name.

• A namespace allows us to disambiguate element names.
  • associate each namespace with a URI (universal resource identifier) like a Web address, so that it's globally unique
  • specify that a given prefix refers to that namespace
  • prepend the prefix to element names from that namespace

• Example:

```
<university-data xmlns:crs='http://www.harvard.edu/course'
                 xmlns:per='http://www.harvard.edu/person'>
  <crs:course>...<crs:course>
  <per:person>...
    <per:name><per:lastname>Sullivan</per:lastname>...
</per:person>
</university-data>
```
Specifying a Separate Schema

- XML doesn’t require a separate schema.
- However, we still need one if we want computer programs to be able to easily process and validate XML documents!
- We’ll look at two different ways of doing so:
  - a DTD (document type definition)
  - an XMLSchema document
- The resulting schema:
  - are less flexible than semistructured-data models
  - can still be more flexible than schema from the other models that we’ve seen
    - for example, can include optional components

XML DTDs

- A DTD (document type definition) is one way to specify a schema for an XML document.
  - basic structure:
    ```xml
    <!DOCTYPE root-element-name [ element-and-attribute-list-declarations ]>
    ```
  - A DTD can be defined:
    - at the start of a XML document
    - in a separate file whose location is specified in the XML document
DTD Element Declarations

- Syntax 1:
  ```xml
  <!ELEMENT element-name (content-description)>
  ```

  _content-description_ is one of the following:
  1. `#PCDATA`: "parsed character data"; text with no child elements
  2. the name of a child element
  3. a regular expression built using instances of 1 and/or 2

- Regular expression syntax:
  ```
  • comp* the component comp appears 0 or more times
  • comp+ the component comp appears 1 or more times
  • comp? the component comp appears 0 or 1 time
  • comp1, comp2 comp1 followed by comp2
  • comp1|comp2 either comp1 or comp2
  ```

DTD Element Declarations (cont.)

- Examples:
  ```xml
  <!ELEMENT university-data ((course|room|person)*)>
  ```
  • a university-data element contains 0 or more child elements
  • the child elements are either course, room, or person elements, and they may appear in any order

  ```xml
  <!ELEMENT university-data (course+, room*, person*)>
  ```
  • a university-data element contains at least one nested course element, followed by 0 or more room elements, followed by 0 or more person elements

  ```xml
  <!ELEMENT course (name, start, end?)>
  ```
  • meaning:
• Examples (cont.):
  `<!ELEMENT person_name (#PCDATA|(last, first))>`
  • meaning:

• Syntax 2:
  `<!ELEMENT element-name content-category >`

  • `content-category` is one or the following:
    • `EMPTY`: the element is an *empty element* that will use a single tag of the form `<name />`
    • `ANY`: the element can contain any data (no restrictions)
DTD Attribute-List Declarations

• Syntax:
  ```xml
  <!ATTLIST elem-name att1-name att1-type att1-default
                   att2-name att2-type att2-default
                   ...
  >
  ```

• Attribute types include the following:
  • CDATA    character data
  • (val1 | val2 | ...) an enumeration of possible values
  • ID       an identifier that must be unique within the document
             (among all ID attributes – not just this attribute)
  • IDREF    has a value that is the value of an ID attribute
             elsewhere in the document
  • IDREFS   a list of ID values from elsewhere in the document

DTD Attribute-List Declarations (cont.)

• Syntax:
  ```xml
  <!ATTLIST elem-name att1-name att1-type att1-default
                   att2-name att2-type att2-default
                   ...
  >
  ```

• Attribute-default specifications include the following:
  • #REQUIRED the attribute must always be specified
  • #IMPLIED the attribute is optional and has no
             default value (omit it if it is not specified)
  • "default-val" the attribute is optional;
               if it isn’t specified, use this default value
Capturing Connections Between Entities

• ID and IDREF attributes can be used to connect elements:

```xml
<course cid="20119" teacher="123456">
  <name>CSCI E-119</name>
  ...
</course>

<course cid="20268" teacher="123456">
  <name>CSCI E-268</name>
  ...
</course>

<person id="123456" teaches="20119 20268">
  <name><last>Sullivan</last><first>David</first></name>
</person>
```

• Allow XML to capture a semistructured graph that isn't a tree.

DTD for the University-Data Domain

```xml
<!DOCTYPE university-data [
  <!ELEMENT university-data ((course|room|person)*)>
  <!ELEMENT course (cname, start, end)>
  <!ATTLIST course
cid ID #REQUIRED
teacher IDREF #REQUIRED
room IDREF #IMPLIED>[
  <!ELEMENT cname (#PCDATA)>
  <!ELEMENT start (#PCDATA)>
  <!ELEMENT end (#PCDATA)>
  <!ELEMENT room (building, room_num?)>
  <!ATTLIST room
rid ID #REQUIRED>[
  <!ELEMENT building (#PCDATA)>
  <!ELEMENT room_num (#PCDATA)>
  <!ELEMENT person (pname, department*)>
  <!ATTLIST person
pid ID #REQUIRED
teaches IDREFS #IMPLIED>[
  <!ELEMENT pname (last, first)>
  <!ELEMENT last (#PCDATA)>
  <!ELEMENT first (#PCDATA)>
  <!ELEMENT department (#PCDATA)>
]>
```
Limitations of DTDs

• ID and IDREF attributes cannot be restricted to identifiers for a specific type of element.
  • why is this problematic?
  •
  •

• More generally, DTDs have an extremely limited type system.
  • examples:
    • can’t specify that an attribute should be an integer
    • can’t specify that all person IDs should have 6 chars

• They don’t allow you to specify namespaces.
  • as a result, element/attribute names in a DTD must be unique
    • example: you can’t have two different name elements – one for people, and one for courses

Limitations of DTDs (cont.)

• They have a limited ability to specify unordered child elements.
  • the | operator allows us to specify some types of unordered collections
    <ELEMENT university-data ((course|room|person)*)>
  • it doesn’t allow us to specify collections in which:
    • all elements must appear at least once
    • we don’t care about the order
    • example: "a course must have a name, a start time, and an end time, and their order doesn’t matter"

• They are written in a format that is different from XML.
  • we need separate tools/algorithms to parse them
**XMLSchema**

- An alternative approach to schema specification that addresses the limitations of DTDs.

- Uses one XML document (*a schema document*) to specify the schema that other XML documents (*instance documents of that schema*) should follow.

- Schema documents are specified using a schema vocabulary.

- The namespace for the XMLSchema vocabulary is associated with the following URL: [http://www.w3.org/2001/XMLSchema](http://www.w3.org/2001/XMLSchema).
  - any prefix can be associated with this namespace, but typically it is either `xsd` or `xs` is used (we'll use `xs`)

- We specify the location of the schema document in the root element of the instance document.

---

**Data Types in XMLSchema**

- *Simple types*: used to specify the type of a single value.
  - examples: `string`, `integer`, `float`, `double`, `boolean`, `date`, `time`, `ID`, `IDREF`, `IDREFS`
  - used for:
    - attributes
    - elements with no attributes and no child elements
      - how would these be specified in a DTD?

- *Complex types*: used to specify the type of an element that has attributes and/or child elements.

- *User-defined types* that are derived from the built-in types.
Specifying a Simple-Type Element or Attribute

- Syntax for specifying a simple-type element:
  \[
  \text{<xs:element name="element-name" type="xs:element-type" />} \\
  \text{examples:} \\
  \text{<xs:element name="building" type="xs:string" />} \\
  \text{<xs:element name="room_num" type="xs:integer" />} \\
  \]

- Syntax for specifying an attribute:
  \[
  \text{<xs:attribute name="attr-name" type="xs:attr-type"} \\
  \text{default = "default-val" use="use-specification" />} \\
  \text{examples:} \\
  \text{<xs:attribute name="rid" type="xs:ID" use="required" />} \\
  \text{<xs:attribute name="color" type="xs:string"} \\
  \text{default = "black" />} \\
  \]

- The default and use attributes are optional

Specifying Elements with Child Elements

- Syntax for an element with child elements:
  \[
  \text{<xs:element name="element-name"} \\
  \text{<xs:complexType} \\
  \text{<xs:compositor} \\
  \text{child element declarations} \\
  \text{attribute declarations} \\
  \text{</xs:compositor} \\
  \text{</xs:complexType} \\
  \text{</xs:element>} \\
  \]

- Example:
  \[
  \text{<xs:element name="room"} \\
  \text{<xs:complexType} \\
  \text{<xs:sequence} \\
  \text{<xs:element name="building" type="xs:string" />} \\
  \text{<xs:element name="room_num" type="xs:integer" />} \\
  \text{</xs:sequence} \\
  \text{<xs:attribute name="rid" type="xs:ID"} \\
  \text{use="required" />} \\
  \text{</xs:complexType} \\
  \text{</xs:element>} \\
  \]
Specifying Elements with Child Elements (cont.)

- Compositors include the following:
  - **sequence**: the elements must appear in the specified order
  - **choice**: one of the specified elements must appear
  - **all**: the specified elements must each appear once, but in any order
    - see the next slide for how to change the required number of occurrences
  - **example**:
    ```xml
    <xs:element name="person">
        <xs:complexType>
            <xs:all>
                <xs:element name="last" type="xs:string" />
                <xs:element name="first" type="xs:string" />
            </xs:all>
        </xs:complexType>
    </xs:element>
    ```

Specifying Elements with Child Elements (cont.)

- You can also specify the number of times that a given element (or a compositor applied to a set of elements) should appear.
  - **syntax**: `<xs:element ... minOccurs="min" maxOccurs="max" >`
    - if `min` is 0, the element is optional
    - for an unlimited # of occurrences, use `maxOccurs="unbounded"`
Named Types

- You can give a name to a type and then use that type elsewhere in the schema document.
  - example:
    ```xml
    <xs:complexType name="personName">
    <xs:all>
    <xs:element name="last" type="xs:string" />
    <xs:element name="first" type="xs:string" />
    </xs:all>
    </xs:complexType>
    ...
    <xs:element name="name" type="personName" />
    ```
  - why isn’t the type of the name element ‘xs:personName’?

Restricted Types

- You can also derive a new type that is based on an existing type.
- One way to do so is to apply a restriction to the values that an existing simple type can take on. Syntax:
  ```xml
  <xs:simpleType name="type-name">
  <xs:restriction base="existing-type">
  restriction declarations
  </xs:restriction>
  </xs:simpleType>
  ```
- Example: specifying that an ID must have a certain length.
  ```xml
  <xs:simpleType name="courseID">
  <xs:restriction base="xs:ID">
  <xs:length value="6" />
  </xs:restriction>
  </xs:simpleType>
  ```
Restricted Types (cont.)

- Other types of restrictions include:
  - `minLength/maxLength`: specify the min/max # of characters
  - `minInclusive`: specify that a number is $\geq$ some value
  - `minExclusive`: specify that a number is $>$ some value
  - `maxInclusive`: specify that a number is $\leq$ some value
  - `maxExclusive`: specify that a number is $<$ some value
  - `enumeration`: specify possible values for this type
  - `pattern`: specify that a string should match a regular expression

- They all follow this syntax:
  
  ```xml
  <xs:restriction-type value="value" />
  ```

List Types

- You can also derive a new type that is list of values of an existing type.
  - syntax:
    
    ```xml
    <xs:simpleType name="list-type-name">
      <xs:list itemType="existing-type-name" />
    </xs:simpleType>
    ```
  
  - example: the definition of the `xs:IDREFS` type
    
    ```xml
    <xs:simpleType name="IDREFS">
      <xs:list itemType="IDREF" />
    </xs:simpleType>
    ```
Using User-Defined Types for Keys

• Recall one of the limitations of DTDs: ID and IDREF attributes cannot be restricted to identifiers for a specific type of element.

• We can employ user-defined types to get around this problem.
  • restrict the values of the ID type
    • ex: make person IDs begin with the letter P followed by 6 digits
      
      \[<xs:simpleType name="personID">\]
      \[<xs:restriction base="xs:ID">\]
      \[<xs:pattern value="P\{0-9\}\{6\}" />\]
      \[</xs:restriction>\]
      \[</xs:simpleType>\]
      
  • similarly restrict the values of the corresponding IDREF type
    
    \[<xs:simpleType name="personIDREF">\]
    \[<xs:restriction base="xs:IDREF">\]
    \[<xs:pattern value="P\{0-9\}\{6\}" />\]
    \[</xs:restriction>\]
    \[</xs:simpleType>\]

  (note: xs:ID values must start with a letter)

Using User-Defined Types for Keys (cont.)

• Continuing the example from the previous slide:
  • if you need a list of IDREFs, create a list type:
    
    \[<xs:simpleType name="personIDREFS">\]
    \[<xs:list itemType="personIDREF" />\]
    \[</xs:simpleType>\]
XPath Expressions

- Used to specify part of an XML document.
  - treat an XML document as a tree
  - specify a path to the relevant nodes in the tree
    - like a pathname in a hierarchical filesystem
- Can be used when performing queries on an XML database.
- **Absolute** path expressions – those that begin with a slash (/) – specify a path that begins at the root of the document.
  - example: `/university-data/course` selects all `course` elements that are children of the `university-data` root element
- Expressions that begin with a double slash (//) select elements from anywhere in the document.
  - example: `//name` selects all `name` 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
  ```
Predicates in XPath Expressions

- Used to select elements that meet a specified condition.
  - surrounded by square brackets
  - applied to the elements selected by the path expression that precedes them.

- Examples:
  - `//course[@teacher="123456"]`
    selects all `course` elements taught by the teacher with ID 123456, where `teacher` is an attribute of `course`
  - `//course[start="19:35"]/name`
    selects the `name` elements of all `course` elements that have a start time of 19:35, where `start` is a child element of `course`
  - `//course[@room]`
    selects all `course` elements with a specified `room` attribute

Predicates in XPath Expressions (cont.)

- Use: `..` to represent a node selected by the preceding path
  - `. . .` to represent the parent of a selected node

- Examples:
  - `//room/room_num[. > 200]`
    selects all `room_num` elements with values > 200
  - `//room[room_num > 200]`
    selects all `room` elements with `room_num` values > 200
  - `//room_num[../building="Sever"]`
    selects all `room_num` elements for rooms located in Sever
      - how could these same elements be selected w/o using `. . .`?

  - note: the original expression would produce extra results if there were other types of elements (other than `room`) with `building` and `room_num` child elements
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-expression_f1,
  $fvar2 IN XPath-expression_f2, …
  LET $lvar1 := XPath-expression_l1, …
  WHERE condition
  ORDER BY XPath-expression_o1, …
  RETURN result-format

- simple example:
  FOR $c in //course
  WHERE $c/start = "19:35"
  RETURN $c/name

FOR Clause

- Like the FROM clause in SQL.

- The query iterates over all combinations of values from the component XPath expressions

  FOR $fvar1 IN XPath-expression_f1,
  $fvar2 IN XPath-expression_f2, …

- $fvar1, $fvar2, etc. are assigned one value at a time

- The WHERE and RETURN clauses are applied to each combination.
RETURN Clause

• Like the SELECT clause in SQL.

• Can be used to perform something like a projection of existing elements and/or attributes:

  ```xml
  FOR $c in //course
  WHERE $c/start = "19:35"
  RETURN $c/name
  ```

• To return a concatenated sequence of items for each value produced by the FOR clause:
  • separate them using a comma
  • surround them with parentheses, because the comma operator has higher precedence
  • example:

  ```xml
  FOR $c in //course
  WHERE $c/start = "19:35"
  RETURN ($c/name, " ", $c/start)
  ```

  • you need to explicitly specify any whitespace between items

RETURN Clause (cont.)

• Can also reshape the output by constructing new elements:

  ```xml
  for $d in document("depts.xml")/depts/deptno
  let $e := document("emps.xml")/emps/emp[deptno = $d]
  ...
  return <big-dept>
  {
    $d,
    <headcount>{count($e)}</headcount>,
    <avgsal>{avg($e/salary)}</avgsal>
  }
</big-dept>
  ```

• When a new element is constructed, we use braces to surround expressions that we want to be evaluated.
  • otherwise, they'll be treated as text that is the value of the new element

• Here again, we use commas to separate items in a concatenated sequence of items.
LET Clause

- Applied to each combination of values produced by the FOR clause.

- Assigns to a variable the *complete set of values* produced by the corresponding XPath expression.
  - unlike a FOR clause, which assigns the results of the XPath expression one value at a time

FOR vs. LET

- Here’s an example that illustrates how they differ:

```xml
for $d in document("depts.xml")/depts/deptno
let $e := document("emps.xml")/emps/emp[deptno = $d]
where count($e) >= 10
order by avg($e/salary) descending
return <big-dept>
  {$d,
   <headcount>{count($e)}</headcount>,
   <avgsal>{avg($e/salary)}</avgsal>
  }
</big-dept>
```
Nested Queries

- We can nest FLWOR expressions:
  - example: create XML elements that group together the person info. for a given instructor with the courses taught by him/her.

```xml
FOR $p$ in /university-data/person[@teaches]
RETURN <instructor-courses>
{ $p,
  FOR $c$ in /university-data/course
  WHERE $c/@cid = $p/@teaches
  RETURN $c
}
</instructor-courses>
```

- result:

```
<instructor-courses>
  <person id="123456" teaches="20119 20268">
    <name><last>Sullivan</last>…</name>
  </person>
  <course cid="20119" teacher="123456">
    <name>CSCI E-119</name> …
  </course>
…
</instructor-courses>
```

Using XQuery for Data Exchange

- The `text()` function gives just the value of a simple element.

- This allows us to use an XQuery FLWOR expression to change the tags used for a given type of data:

```xml
FOR $p$ in /university-data/person[@teaches]
RETURN <instructor>
  {<name>{$p/name/first/text(), " ", $p/name/last/text()}</name>,
   FOR $c$ in /university-data/course
   WHERE $c/@cid = $p/@teaches
   RETURN <course>{$c/name/text()}</course>
  }
</instructor>
```

- result:

```
<instructor>
  <name>David Sullivan</name>
  <course>CSCI E-119</course>
  <course>CSCI E-268</course>
…
</instructor>
```
Using XQuery for Data Display

• A FLWOR expression can be used to generate web pages that display the contents of a database:

```xml
<html>
<body>
<h2>Faculty and the Courses They Teach</h2>
<ul>
  FOR $p in /university-data/person[@teaches]
  RETURN <li>({<b>{$p/name/first/text(), " ", $p/name/last/text()}</b>,
  <ul>
    FOR $c in /university-data/course
    WHERE $c/@cid = $p/@teaches
    RETURN <li>{$c/name/text()}</li>
  </ul>
  }</li>
</ul>
</body></html>
```

XML-Enabled DBMS

• A DBMS that is based on another model, but has the ability to export and import XML.
  • the XML is not necessarily "visible" in the stored data
  • publish: convert data from another model to XML
  • shred: convert XML data to another data model

• An appropriate logical-to-logical mapping is needed.
  • could derive it from a user-supplied XML schema
Native XML DBMS

• Stores XML documents in a form based on the XML data model.
  • the XML is "visible" in the stored data
    (although not necessarily as it would appear in a text file)

• Can store any XML document.
  • including those with no separate schema

• Two approaches to implementing one:
  1. build it on top of a DBMS that uses another model
     • use a fixed (non-user-defined) logical-to-logical mapping
       that can accommodate any XML document
  2. build it directly on top of a storage engine
     • define the appropriate logical-to-physical mapping

Native XML DBMS: Logical-to-Logical Mappings

• Possible XML-to-relational mappings:
  1) use a relational schema that stores an entire XML document
     as the value of a text attribute:
       document(id, docstring)
       • useful if you need to preserve the exact bytes of the
         original document (ex: for legal purposes)
       • 100% round-tripping: what goes in is what comes out
       • may also be useful if you have small documents that are
         typically retrieved in their entirety
  2) use a relational schema that encodes the tree structure
     of the document:
       document(id, name)
       element(doc_id, elem_id, parent_id, name)
       attribute(doc_id, attr_id, parent_id, name)
       text(doc_id, text_id, parent_id, value)
Logical-to-Logical Mappings (cont.)

• Example of mapping #2 from the previous slide (omitting doc_id):

```xml
<univ-data>
  <course>
    <name>CSCI E-268</name>
  </course>
  <person>
    <name><last>Sullivan</last><first>David</first></name>
  </person>
</univ-data>
```

<table>
<thead>
<tr>
<th>elem_id</th>
<th>parent_id</th>
<th>name</th>
<th>text_id</th>
<th>parent_id</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>null</td>
<td>univ-data</td>
<td>500</td>
<td>102</td>
<td>CSCI E-268</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
<td>course</td>
<td>501</td>
<td>110</td>
<td>Sullivan</td>
</tr>
<tr>
<td>102</td>
<td>101</td>
<td>name</td>
<td>502</td>
<td>111</td>
<td>David</td>
</tr>
<tr>
<td>108</td>
<td>100</td>
<td>person</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>108</td>
<td>name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>109</td>
<td>last</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>109</td>
<td>first</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Native XML DBMS: 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
Berkeley DB XML

- A native XML database system built on top of Berkeley DB
- Includes support for queries usingXPath/XQuery.
- As in most native XML databases, the fundamental unit of storage is a document.
  - a root element, and possibly some number of nested elements
  - treated like a row/tuple in a relational DB
  - what do you think this would this map to in BDB?
- Related documents are grouped into a container.
  - comparable to a table in a relational DB
  - what do you think this would this map to in BDB?

BDB XML (cont.)

- Example: you could have a container named contacts.dbxml that contains documents that look something like this:
  
  `<contact>
  <name> <last>Codd</last>
  <first>Ted</first>
  </name>
  <phone type="home">123-456-6789</phone>
  </contact>
  
- Different documents in a given container can have different schema.
Logical-to-Physical Mapping in BDB XML

- Each container is stored in its own BDB file.

- A container’s BDB file includes multiple databases:
  - one that stores the contents of the documents
  - others that represent indices maintained on the documents
    - important for good query performance
  - dictionaries that map element and attribute names to numeric codes
    - these codes are used in the indices

Logical-to-Physical Mapping in BDB XML (cont.)

- Two types of containers:
  1) wholedoc containers: store entire, intact documents
    - like the first XML-to-relational mapping
    - key-data pairs:
      - key = doc_id
      - data = document as text
  2) node containers: store XML documents as collections of nodes – where each node is an element
    - key-data pairs:
      - key = doc_id
      - data = a marshalled element, or document metadata
Distributed Databases and Replication

Computer Science 460/660
Boston University
Fall 2013
David G. Sullivan, Ph.D.

What Is a Distributed Database?

• One in which data is:
  • partitioned or fragmented among several sites and/or
  • replicated – copies of the same data are made available at multiple sites

• It is managed by a distributed DBMS (or DDBMS) – DBMSs at two or more sites that jointly provide access to a single logical database.
What Is a Distributed Database? (cont.)

- Example:

```
  site 1       site 2
  DB          DB
  site 3       site 4
  network
```

- 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

- Divides up the records in a database among several sites.
  - the resulting "pieces" of the database are known as fragments

- Let \( R \) be a collection of records of the same type (e.g., a relation).

- **Horizontal fragmentation** divides up the "rows" of \( R \).
  - \( R(a, b, c) \rightarrow R1(a, b, c), R2(a, b, c), \ldots \)
  - \( R = R1 \cup R2 \cup \ldots \)

- **Vertical fragmentation** divides up the "columns" of \( R \).
  - \( R(a, b, c) \rightarrow R1(a, b), R2(a, c), \ldots \) (a is the primary key)
  - \( R = R1 \Join R2 \Join \ldots \)
Fragmentation (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:

<table>
<thead>
<tr>
<th>account</th>
<th>owner</th>
<th>street</th>
<th>city</th>
<th>branch</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111</td>
<td>E. Scrooge</td>
<td>1 Rich St</td>
<td>...</td>
<td>main</td>
<td>$11111</td>
</tr>
<tr>
<td>123456</td>
<td>R. Cratchit</td>
<td>5 Poor Ln</td>
<td>...</td>
<td>west</td>
<td>$10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>222222</td>
<td>west</td>
<td>$10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>444444</td>
<td>south</td>
<td>$70000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Here’s one way of fragmenting it:
Replication

- Replication involves putting copies of the same collection of records at different sites.

<table>
<thead>
<tr>
<th>account type</th>
<th>interest rate</th>
<th>monthly fee</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>0%</td>
<td>$10</td>
</tr>
<tr>
<td>big saver</td>
<td>2%</td>
<td>$50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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
  - need to still provide the ACID properties
  - atomicity and isolation can be harder to guarantee

Important Goal: Transparency

- Users shouldn't need to know:
  - that a database has been fragmented
  - where the fragments are located

- Instead, they are able to treat the database as a single logical database that is fully available at all of the sites.

- In addition, users shouldn't need to know about replication.
  - send their requests and updates to one site
  - they will be propagated to the replicas as needed

- It's possible to have a limited form of transparency.
  - example:
    - require fragment-specific names in queries
    - don't require knowledge of fragment locations or of the degree of replication
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: occurs if communication failures cause there to be no communication between two sets of sites in the system

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.

- Each site maintains its own logs, recovery system, lock tables, …

- The sites need to coordinate with each other to ensure that either all subtxns commit or all of them abort.
Distributed Atomicity

- To maintain atomicity in a distributed database:
  - before any site commits its subtxn, it must ensure all other sites are able to commit their subtxns
  - if any site is unable to complete its subtxn, all of the sites must abort their subtxns

- Example of what could go wrong:
  - a subtxn at one of the sites deadlocks and is aborted
  - before the coordinator finds out about this, it notifies the other sites that it's time to commit, and they commit

- 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)

- Two-phase commit is a protocol for deciding whether to commit or abort a distributed transaction.

- Basic idea:
  - coordinator asks sites if they're ready to commit
  - if all sites say they're ready, all subtxns are committed
  - otherwise, all subtxns are aborted

- Before saying that it's ready, a site must ensure that its subtxn can be either committed or aborted – even after a failure.
  - what steps would be needed?

  - after these steps, the subtxn is in the ready state
    - aka the precommitted state
    - a site in the ready state 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 decides that it is ready to 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 decides that it 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 a site fails to send a message within some time interval, the coordinator 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 Phase II: Commit or Abort (cont.)

- Many implementations of 2PC include the following additions:
  - a site acknowledges its receipt of the commit/abort message by sending the coordinator an *ack message*
  - once the coordinator receives acks from all sites, it writes an end log record for the transaction

- The acks allow the coordinator to know when it can remove info. about a txn from its in-memory transaction table.

- These additions also allow the coordinator to know whether it should resend the commit/abort messages after recovering from a crash.

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

• The decision of whether to undo or redo a distributedtxn T is based on the last record for T in the log.

• If the last log record for T is anything but a ready record: treat T like any other txn.
  • if it's a commit record, redo T's updates as needed
  • otherwise, undo T's updates as needed
    • why does this work if the last record is do-not-commit?

  • what if the last record is from before 2PC began?

Recovery When Using 2PC (cont.)

• If the last log record for T is a ready record: contact the coordinator (or another site) to determine T's fate.
  • because the log records for T were forced to disk, the site can still commit or abort T as needed
  • if it can't reach another site, it must block until it can find out what to do!
What if the Coordinator Fails?

• The other sites can either:
  • wait for the coordinator to recover
  • elect a new coordinator

• If a participating site received a prepare message for a txn T from the failed coordinator but hadn’t yet sent back a ready message, it can abort T now.
  • why is this always safe?

  • this is preferable to trying to continue with the protocol, because it allows the fate of T to be decided

What if the Coordinator Fails? (cont.)

• If a participating site sent a ready message for T to the failed coordinator but didn’t get back a commit or abort message, it can poll the other surviving sites about T’s fate:
  • if at least one site has a commit record for T, T was committed
  • if at least one site has an abort record for T, T was aborted
  • if no site has either a commit or abort record and at least one site does not have a ready record for T, T can be aborted now
  • if no site has either a commit or abort record, and all surviving sites have ready records for T, can’t determine T’s fate unless the failed coordinator recovers!
    • why?
2PC Example

- T requires subtxns at sites A, B, C. Site A is the coordinator.
- After the last operation in T, site A:
  - sends prepare messages to B and C
  - puts its own subtxn in the ready state.
- Sites B and C:
  - put their subtxns in the ready state
  - send ready messages to A
- Site A:
  - writes its commit record (T is now officially committed)
  - sends commit messages to B and C
- B writes its commit record.
- C crashes before writing its commit record. What happens?

Types of Replication

- In synchronous replication, transactions are guaranteed to see the most up-to-date value of an item.
- Because synchronous replication is often too slow, asynchronous replication is often used in practice.
  - transactions are not guaranteed to see the latest 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.
- When reading, read enough copies to ensure you get at least one copy of the most recent value.
  - use a version number to determine which value is most recent
  - the copies "vote" on the value of the item
- Drawback: reads are now more expensive
- 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 \( n - w + 1 \)) -- but see later notes!
- example: 6 replicas
  - update 4
  - must read at least 3 (not necessarily a majority!)
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 follows:
    • each write updates 3 copies
    • each read accesses 4 copies
  • can end up with schedules that are not conflict serializable

• example:

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x(A1); x(A2); x(A3) ) ( w(A1); w(A2); w(A3) )</td>
<td>( x(A4); x(A5); x(A6) ) ( w(A4); w(A5); w(A6) )</td>
</tr>
<tr>
<td>( x(B1); x(B2); x(B3) ) ( w(B1); w(B2); w(B3) )</td>
<td>( x(B4); x(B5); x(B6) ) ( w(B4); w(B5); w(B6) )</td>
</tr>
</tbody>
</table>

Xi = the copy of item X at site i

\( T_1 \) should come before \( T_2 \) based on the order in which they write A.

\( T_1 \) should come after \( T_2 \) based on the order in which they write B.
What Do We Need?

- We need a txn to be able to acquire a lock for a *logical item*, not just for individual copies of that item.
  - this doesn't necessarily mean locking every copy
  - it means preventing someone else from reading and/or writing what is considered the current value of the item according to whatever replication scheme is being used
  - we say that a transaction acquires a *global lock* for the item

- Requirements for global locks:
  - no two transactions 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 take appropriate steps to ensure the proper ordering of operations within a given distributed txn.
  - 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

- In *centralized locking*, one site manages the lock requests for all items in the distributed database.
  - even items that do not have copies stored at that site
  - since there is only one place to acquire locks, these locks are obviously global locks

- Problems with this approach?

Option 2: Primary-Copy Locking

- *Primary-copy locking* is a variation on centralized locking.

- To prevent one site from becoming a bottleneck, we divide up the responsibility for managing lock requests among the sites in the DDBMS.

- One copy of an item is designated the *primary copy* of that item.

- The site on which the primary copy of an item resides is responsible for handling all lock requests on that item.
  - locking the primary copy of an item gives you a global lock on the item
Option 3: Fully Distributed Locking

- In *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 a global lock

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:

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>xl(A_1); xl(A_2); xl(A_3)</td>
<td>xl(A_4); xl(A_5); xl(A_6)</td>
</tr>
<tr>
<td>w(A_1); w(A_2); w(A_3)</td>
<td>w(A_4); w(A_5); w(A_6)</td>
</tr>
<tr>
<td>xl(B_1); xl(B_2); xl(B_3)</td>
<td>xl(B_4); xl(B_5); xl(B_6)</td>
</tr>
<tr>
<td>w(B_1); w(B_2); w(B_3)</td>
<td>w(B_4); w(B_5); w(B_6)</td>
</tr>
</tbody>
</table>

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
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:
      ![Diagram showing a scenario with two sites and transactions T1 and T2]
      site 1: T1 → T2
      site 2: T1 ← T2
    • one option: periodically send local waits-for graphs to one site that is responsible for performing global deadlock detection

• Instead of using deadlock detection, it is often easier to use the timeout-based scheme that we covered earlier.
  • if a txn waits for too long, presume deadlock and abort it

Asynchronous Replication I: Primary Site

• In primary-site replication, one replica is designated the primary or master replica.
  • Only the master is updatable.
  • The other replicas (the secondaries or clients) can only be read.
    • no locks are acquired when accessing the clients
    • thus, we only use them when performing read-only txns
  • Changes to the master are periodically propagated to the clients.
    • need to ensure they are applied in such a way that you always have a consistent snapshot

• Drawbacks of this approach?
Asynchronous Replication II: Peer-to-Peer

- In *peer-to-peer replication*, more than one replica is updatable.
- Problem: need to somehow resolve conflicting updates!
Performance Tuning

Computer Science 460/660
Boston University
Fall 2013
David G. Sullivan, Ph.D.

Performance Tuning in Database Systems

- Topics for this unit:
  - goals and challenges of tuning
  - what can be tuned: three levels
  - deciding what to tune
  - example tuning scenarios
  - a brief look at automated tuning

- Good 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, ops/sec
  • measure over some interval (time-based or work-based)

• Decrease *response time* or *latency* = time spent waiting for an operation to complete
  • overall throughput may be good, but one or more types of transactions 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.
  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
  • may encounter workloads you haven’t seen before
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
Memory or Disks?

• If you have a fixed amount of money to spend on disks and memory, how should you spend it?

• Determining disk costs (rough approximation):
  • variables:
    • $p_D$ = price of one disk
    • $a$ = # of accesses per second that each disk can support
  • to support $k$ accesses/sec, need $\frac{k}{a}$ disks
  • total cost = $\frac{k}{a} p_D = k \frac{p_D}{a}$
    • cost of supporting 1 access/sec
  • if a page is accessed on disk $n$ times per sec, it costs $n \frac{p_D}{a}$
    (note: $n$ may be a fraction < 1)

• Determining memory costs:
  • variables: $p_M$ = price of 1 MB
    • $m$ = number of pages per MB
  • storing a page in memory costs $\frac{p_M}{m}$

Memory or Disks? (cont.)

• Break-even point: disk cost = memory cost
  • $n \frac{p_D}{a} = \frac{p_M}{m}$
  • solve for $n$.
  • if a page is accessed at least $n$ times per sec, it’s cost-effective to cache it in memory.

• 5-minute rule for randomly accessed pages:
  • Gray et al. (1997) get $n = \frac{1}{266}$ (i.e., 1 access per 266 sec)
  • 266 sec $\approx$ 5 minutes
  • rule of thumb = buy enough memory to cache all pages that are accessed at least once every 5 minutes
  • 1-minute rule for pages that are accessed sequentially

• Gray et al. also show that $n$ stayed roughly the same from 1987 to 1997
  • despite significant changes in the relevant values
Other Details of Hardware Tuning

- Can also add:
  - processing power
  - network bandwidth (in the case of a client-server 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:
    - give the log its own disk
    - increase the amount of prefetching, etc.

2. Parameter Tuning (Middle Level)

- DBMSs—like most complex software systems—include parameters ("knobs") that can be tuned by the user.

- Example knobs:
  - page size – see later
  - approach to handling deadlocks (detection, wait-die, etc.)
  - deadlock-resolution algorithm
  - deadlock-detection interval
  - checkpoint interval
  - multiprogramming level (MPL) – see later
  - locking granularity – see later
  - locking behavior (e.g., DB_RMW in Berkeley DB)

- 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

- How could a smaller page size improve performance?

```
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

8K page

Tuning Page Size (cont.)

- Disadvantages of a smaller page size:
  - can lead to more overflow pages
  - if page size < filesystem block size, may need to read a file block before writing a page

```
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
```

- What if we select page size > block size? (assume a multiple of the block size)

- Rule of thumb?
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 partitioning: 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?
Tuning Indices

- If SELECTs are slow, add one or more index.
- If UPDATEs 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 clustered/internal?
- 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: the long, read-only txns operate on a snapshot of the database and do not conflict with the short update txns
  - use a weaker isolation level for the long txns
    - ex: READ COMMITTED – don't hold shared locks till end
    - ex: READ UNCOMMITTED – read-only txns acquire no 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!
  endif
Monitoring the System

- Examine performance stats, including:
  - hit rate: % of accesses satisfied in the cache
    - possibly two caches: DBMS cache and OS buffer cache
    - get buffer-cache hit rate from OS (e.g., if virtual memory and buffer cache are integrated, look at page-fault stats)
  - processor usage
  - disk usage
  - lock waits and aborts

- Can usually obtain both:
  - overall stats
  - stats for particular txns/queries

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
Automated Tuning

“It is...black magic...how databases are tuned. It is tough to ship a tuning guru with every database.”
Surajit Chaudhuri, Microsoft Research

• Because manual tuning is hard, many have proposed making systems self-tuning.
  • Asilomar Report on Database Research (1998)

• The need for automated tuning continues to increase.
  • software systems are increasingly complex
  • individuals & small organizations can’t afford a guru
  • embedded software systems must be self-tuning

What Has Been Done?

• Commercial DBMSs have some degree of self-tuning.
  • SQL Server uses feedback to adjust cache size
  • DB2 has a “configuration wizard” that configures the sizes of various memory regions for a given workload (appears to be model-based)
  • several systems have wizards to select indices and views for a particular workload (also model-based)

• Numerous research efforts:
  • Weikum et al. (1994) use feedback to adjust MPL
  • Brown et al. (1994-96) use feedback to adjust sizes of memory regions and MPL
  • automated tuning of other types of software systems
  • Sullivan developed a methodology for automated tuning based on probabilistic reasoning (2003)
  • still an open research area...