Midterm Notes

What is a Database Management System (DBMS)?

- Massive
  - A large amount of data is being managed (i.e.2 terabytes)
  - Data cannot be stored in main memory, stored on physical disks somewhere
- Persistent
  - Data need to be available in a consistent state
- Convenient
  - SQL allows for dynamic on-the-fly searches
- Efficient
  - The system understands the language and comes up with the right way of executing queries efficiently (query optimization)
  - Standard benchmark (TPC) that estimates how fast the database system runs
- Safe
  - Safety from system failure
    - If a web server crashes, or even if the database server crashes, the data is still safe and unharmed
  - Safety from malicious users
    - The system shows the right sets of the data to the right users
- Multi-User
  - Example: A husband and wife have $300 in their bank account. John withdrawals $100 and Susan withdrawals $50 at different locations at the same time. What is the remaining balance left in the account?
- Location of data – In concurrent programming, the data is in memory, where as in DBMS, the data is in a physical disk
- Granularity of data – In a file system, the granularity of data is a file/page, in DBMS, the granularity of data is in tuples

Database construction steps

- Domain analysis
- Database design
  - Entity-Relational model – Analyze the relations in the database
- Table creation
  - Data Definition Language (DDL)
- Data load
  - No standard SQL command to load bulk data
- Query & update
  - Data Manipulation Language (DML)

Relational algebra

- High level concept
  - Relational model stores tables into the disk
  - Generates relations based on the query that is executed upon the disk
- Query – a question
  - Poses a query to the underline system, and the system returns an answer
- Operators
  - Selection operator "σ": Returns the tuples in the table where the tuples meets the requirements specified by the operator
    - i.e., *Find all students that are younger than 18* ≡ $\sigma_{age<18}$(Student)
    - i.e., *Find all students that are younger than 18 and has a GPA greater than 3.7* ≡ $\sigma_{age<18 \wedge GPA > 3.7}$(Student)
    - Valid Boolean operators: $>, <, =, \neq, \geq, \leq, \wedge, \vee, \neg$
  - Projection operator "∏": Returns the chosen attributes in a table
    - i.e., *Find the department of all courses* ≡ π_{dept}(Course)
    - i.e., *Find the name, address, and GPA of all students* ≡ $\pi_{name, addr, GPA}$(Student)
  - Cross-product operator "×": Returns all possible combinations of the rows in two tables
    - i.e., *Find all combinations of students and enrolled students* ≡ Student × Enroll = (Student.sid, name, addr, age, GPA, Enroll.sid, dept, cnum, sec)
    - Number of resulting tuples is $|table_1| * |table_2|$ (i.e., 4 * 4 = 16)
    - Conflicts in attribute names are prefaced by the table name
  - (Natural) Join operator "⋈": Returns the cross-products of table tables where common attributes match in value
    - i.e., *Find all the enrolled students and the course that they are enrolled in* ≡ Enroll ⋈ Course ≡ $\sigma_{Enroll.dept = Ccourse.dept \wedge Enroll.cnum = Course.cnum \wedge Enroll.sec = Course.sec}$(Enroll × Course)
  - θ-join operator "$\bowtie_\theta$": Returns tuples in a cross-product that meets the selection criteria
    - i.e., *Find all combinations of students with a GPA higher than 3.7 and enrolled students* ≡ Student $\bowtie_{Student.sid = Enroll.sid \wedge GPA > 3.7}$ Enroll ≡ $\sigma_{Student.sid = Enroll.sid \wedge GPA > 3.7}$(Student × Enroll)
  - Set union operator "∪": Returns the union of two tables
    - Student(sid, name, addr, age, GPA)
      GradStudent(sid, name, addr, age, GPA, advisor)
    - i.e., *Find the SIDs of all the students* ≡ Π_{sid}(Student) ∪ Π_{sid}(GradStudent)
  - Set difference operator "–": Return the difference of two tables
    - i.e., *Find the department, course number, and section of all classes without enrolled students* ≡ $\Pi_{dept, cnum, sec}$(Class) – $\Pi_{dept, cnum, sec}$(Enroll)
    - i.e., *Find the titles of all the classes without enrolled students* ≡ Π_{title}(($\Pi_{dept, cnum, sec}$(Class) – $\Pi_{dept, cnum, sec}$(Enroll)) ⋈ Class)
  - Set intercept operator "∩": Returns the interception of two tables
    - i.e., *Find all the instructors who teaches both CS and EE classes* ≡ $\Pi_{instructor}(\sigma_{dept='CS'}$(Class)) ∩ $\Pi_{instructor}(\sigma_{dept='EE'}$(Class))
    - R ∩ S = R – (R – S)
  - Rename operator "ρ": Renames a table to another name
    - i.e., *Find pairs of student names who live at the same address* ≡ $\sigma_{R.name > S.name}(\Pi_{name, name}(\sigma_{R.addr = S.addr}(\rho_R(Student) \times \rho_S(Student))))$
  - Division operator "/": Returns tuples that exist in all of another tuple/table
    - i.e., *Find the SIDs of students who is enrolled in all of the CS classes*
      - Things we'd need to know:
        - ✓ All possible combinations of students taking CS classes ≡ Π_{sid}(Student × $\Pi_{dept, cnum, sec}(\sigma_{dept='CS'}$(Class))) – Enroll
        - ✓ Students who are not taking at least one of the classes ≡ All possible combinations – enrolled students
        - ✓ Students who are taking all CS classes ≡ All students – students who are not taking at least one of the classes
      - Π_{sid}(Student) – Π_{sid}(Π_{sid}(Student × $\Pi_{dept, cnum, sec}(\sigma_{dept='CS'}$(Class))) – Enroll)
    - R/S = $\Pi_A(R) - \Pi_A((\Pi_A(R) \times S) - R)$
  - Common mistake: Questions that need to use the logical opposite approach
    - i.e., *Find the SIDs of students who are not taking any CS classes*
      - $\Pi_{sid}(\sigma_{dept \neq 'CS'}$(Student ⋈ Enroll)) … WRONG!
      - Π_{sid}(Student) – $\Pi_{sid}(\sigma_{dept='CS'}$(Enroll)) … CORRECT!

Standard Query Language (SQL)

- Operators (multi-set or bag semantic):
  - SELECT attribute: Specify the chosen attribute in a table
  - FROM table: Specify the table to execute the query
  - WHERE condition: Specify the conditionto choose tuples
  - ORDER BY attribute [ASC/DESC]: Specify an attribute to order the result by
  - DISTINCT: Eliminate duplicate tuples in result
  - Wildcards "_" and "%" – "_" denotes single character, "%" denotes any number of characters
  - LIKE: Matches substrings
  - GROUP BY: Specify an attribute to group by
  - HAVING: Specify a condition on aggregates to choose by
- Set operators (set semantic):
  - UNION: Returns a table that is the first table and the second table appended together
  - INTERSECT: Returns the tuples in the first table that also exist in the second table
  - DIFFERENCE: Returns the tuples in the first table subtract the second table

    - EXCEPT: Returns the tuples in the first table that is not in the second table
    - NOT IN: Returns true if tuple is not in a specified set
    - ALL: Returns true if tuple compares true to all elements of a set
    - SOME: Returns true if tuple compares true to at least one element of a set
- Subqueries
    - i.e., Give me the names of students who are not taking any CS classes

```
SELECT name
FROM Student
WHERE sid NOT IN
   (SELECT sid
   FROM Enroll
   WHERE dept='CS')
```

- Aggregation operators
    - AVG: Returns the average of the selected attributes, does not average null values
    - COUNT: Returns the count of the selected attributes, counts null values
    - MAX/MIN: Returns the max or min of the selected attributes, does not consider null values
- Database modification operators
    - INSERT INTO table VALUES ($attr_1$, …)
    - INSERT INTO table query
    - UPDATE table SET $attr_1$ = new_value [WHERE condition]
    - DELETE FROM table WHERE condition

<u>Integrity constraints</u>

- Key constraints
    - Defines the keys of each table, which uniquely identifies a tuple
    - If a key is not defined as NOT NULL, then DB2 will give an error prompting the key to be defined as NOT NULL
    - i.e., Course(dept, cnum, sec, unit, instructor, title)

```
CREATE TABLE Course (
   dept CHAR(2) NOT NULL,
   cnum INT NOT NULL,
   sec INT NOT NULL,
   unit INT,
   instructor VARCHAR(30),
   title VARCHAR(30),
   PRIMARY KEY(dept, cnum, sec),
   UNIQUE(dept, cnum, instructor),
   UNIQUE(dept, sec, title) )
```

- Referential integrity
    - Values of one attribute values in one table will always appear in the values of an attribute of another table (i.e., Enroll.sid will always appear in Student.sid)
    - Value of a foreign key can be NULL
    - Operators that may cause violations
        - DELETE from referenced table
        - UPDATE on referenced table
        - INSERT into referencing table
        - UPDATE on referencing table
    - When changing the referencing table, the DBMS may abort the command and not allow the change
    - i.e., sid must be an entry in Student; dept, cnum, sec must be entries in Class

```
CREATE TABLE Enroll (
   sid INT REFERENCES Student(sid),
   dept CHAR(2),
   cnum INT,
   sec INT,
   FOREIGN KEY (dept, cnum, sec)
          REFERENCES Class(dept, cnum, sec)|
                 ON DELETE CASCADE
                 ON UPDATE SET NULL )
```

    - Referencing attributes called FOREIGN KEY
    - Referenced attributes must be PRIMARY KEY or UNIQUE
- Tuple (CHECK) constraint
    - Enforced for every tuple within a particular table
    - CHECK(<condition>) in CREATE TABLE
        - i.e., *The units of all CS classes are above 3*

```
CRATE TABLE Student(
   dept CHAR(2),
   cnum INT,
   unit INT,
   title VARCHAR(50),
   CHECK (dept <> 'CS' OR unit > 3) )
```

    - Constraint is checked whenever a tuple is inserted/updated.
    - In SQL92, conditions can be more complex, e.g., with subqueries
- SQL assertions
    - Constraint on the entire relation or database
    - CREATE ASSERTION <assertion_name> CHECK (<condition>)
    - i.e., Average GPA > 3.0

```
CREATE ASSERTION HighGPA CHECK (
   3.0 < (SELECT AVG(GPA) FROM Student) )
```

<u>Disks</u>

- A typical disk
    - Platter diameter: 1-5 in
    - Cylinders: 100 – 2000
    - Platters: 1 – 20
    - Sectors per track: 200 – 500
    - Sector size: 512 – 50K
    - Overall capacity: 1G – 200GB
        - (sectors / track) × (sector size) × (cylinders) × (2 × number of platters)
- Disk access time
    - Access time = (seek time) + (rotational delay) + (transfer time)
        - Seek time – moving the head to the right track
        - Rotational delay – wait until the right sector comes below the head
        - Transfer time – read/transfer the data
- Seek time
    - Time to move a disk head between tracks
        - Track to track ~ 1ms
        - Average ~ 10 ms
        - Full stroke ~ 20 ms
- Rotational delay
    - Typical disk:
        - 3600 rpm – 15000 rpm
        - Average rotational delay?
            - 3600 rpm / 60 sec = 60 rps; delay = 1/60 sec
- Transfer rate
    - Burst rate
        - (# of bytes per track) / (time to rotate once)
    - Sustained rate
        - Average rate that it takes to transfer the data
        - (# of bytes per track) / (time to rotate once + track-to-track seek time)
- Abstraction by OS
    - Sequential blocks – No need to worry about head, cylinder, sector
    - Access to random blocks – Random I/O
    - Access to consecutive blocks – Sequential I/O
- Random I/O vs. Sequential I/O
    - Assume
        - 10ms seek time
        - 5ms rotational delay
        - 10MB/s transfer rate
        - Access time = (seek time) + (rotational delay) + (transfer time)
    - Random I/O
        - Execute a 2K program – Consisting of 4 random files (512 each)
        - ((10ms) + (5ms) + (512B / 10MB/s)) × 4 files = 60ms
    - Sequential I/O
        - Execute a 200K program – Consisting of a single file
        - (10ms) + (5ms) + (200K / 10MB/s) = 35ms
- Block modification
    - Byte-level modification not allowed
        - Can be modified by blocks
    - Block modification
        - Read the block from disk
        - 2. Modify in memory
        - 3. Write the block to disk
- Buffer, buffer pool
    - Keep disk blocks in main memory
        - Avoid future read
        - Hide disk latency
    - Buffer, buffer pool
        - Dedicated main memory space to "cache" disk blocks
        - Most DBMS let users change buffer pool size

Files

- Spanned vs. Unspanned
  - Unspanned – Store as many tuples into a block, forget about the extra remaining space
  - Spanned – Store as many tuples into a block, store part of the next tuple into the block
- Deletion
  - For now, ignore spanning issue, irrelevant for current discussion
  - What should we do?
    - Copy the last entry into the space
    - Shift all entries forward to fill the space
    - Leave it open and fill it with the next update
      - Have a pointer to point to the first available empty slot
      - Have a bit-map of the occupancy of the tuples
- Variable-Length Tuples
  - Reserved Space – Reserve the maximum space for each tuple
  - Variable Length
    - Tuple length in the beginning
    - End-of-record symbol
    - Pack the tuples tightly into a page
  - Update on Variable Length Tuples?
    - If new tuple is shorter than the tuple before – just place it where it was
    - If new tuple is longer than the tuple before – delete the old tuple and place it at the end of the block with free space
  - Slotted Page
    - Header slots in the beginning, pointing to tuples stored at the end of the block
- Long Tuples
  - Spanning
  - Splitting tuples – Split the attributes of tuples into different blocks
- Sequential File – Tuples are ordered by some attributes (search key)
- Sequencing Tuples
  - Inserting a new tuple
    - Easy case – One tuple has been deleted in the middle
      - Insert new tuple into the block
    - Difficult case – The block is completely full
      - May shift some tuples into the next block, if there are space in the next block
      - If there are no space in the next block, use the overflow page
  - Overflow page
    - Overflow page may over flow as well
      - Use points to point to additional overflow pages
      - May slow down performance, because this uses random access
  - Any problem?
    - PCTFREE in DBMS
      - Keeps a percentage of free space in blocks, to reduce the number of overflow pages
      - Not a SQL standard

Indexing

- Basic idea – Build an "index" on the table
  - An auxiliary structure to help us locate a record given to a "key"
  - Example: User has a key (40), and looks up the information in the table with the key
- Indexes to learn
  - Tree-based index
    - Index sequential file
      - Dense index vs. sparse index
      - Primary index (clustering index) vs. Secondary index (non-clustering index)
    - B+ tree
  - Hash table
    - Static hashing
    - Extensible hashing
- Dense index
  - For every tuple in the table, create an index entry which to search on, and a pointer to the tuple that it points to (so just an index with pointers to the tuple in the block that the tuple is in)
  - Dense index blocks contain more indexes per block than tuples in their blocks, because dense indexes are much smaller in size than the tuple that they point to.
- Why dense index?
  - Example:
    - 1,000,000 records (900-bytes/rec)
    - 4-byte search key, 4-byte pointer
    - 4096-byte block
  - How many blocks for table?
    - Tuples / block = size of block / size of tuples = 4096 / 900 = 4 tuples
    - Records / tuples = 1,000,000 / 4 = 250,000 blocks
    - 250,000 blocks * 4096 bytes / block = 1GB
  - How many blocks for index?
    - Index / block = 4096 / 8 = 512
    - Records / indexes = 1,000,000 / 512 = 1956
    - 1956 blocks * 4096 bytes / block = 8MB
- Sparse index
  - For every block, create an index entry which to search on, and a pointer to the block that it points to (even smaller index size of the dense index)
  - In real world, this reduces the index size dramatically, because there may be many tuples in one block, for which sparse index only creates on index entry to those tuples
- Sparse 2nd level
  - For every index block, create an index entry which to search on, and a pointer to the index block that it points to (an index on the index, which further reduces in size)
  - Can create multiple level of indexes (multi-level index)
- Terms
  - Index sequential file (Index Sequential Access Method)
  - Search key (≠ primary key)
  - Dense index vs. Sparse index
  - Multi-level index
- Duplicate keys
  - Dense index, one way to implement – Create an index entry for each tuple
  - Dense index, typical way – Create an index entry for each unique tuple
- Updates on the index?
  - Insertion (empty) – First follow the link structure to identify where the tuple should be located (found with enough space)
  - Insertion (overflow) – Create an overflow block with a pointer from the original block, which adds the entry 15 into the overflow block
  - Insertion (redistribute)
    - Try to move blocks to other adjacent blocks
    - Update any changes to the indexes as needed
- Deletion (one tuple)
  - See which index block the tuple is located
  - If the first entry of the block is not deleted, no update to index necessary
  - If the first entry of the block is deleted, update the index appropriately
- Deletion (entry block)
  - If the entire block is deleted, the index entry can be deleted
  - Move all index entries within the block up to compact space
- Primary index – Index that is created over the set of attributes that the table is stored (also called clustering index)
- Secondary index
  - Index on a non-search-key
  - Unordered tuples – Non-sequential file
  - Sparse index make sense?
    - Does not make sense because the files are not not in sequence
  - Dense index on first level
  - Sparse index from the second level
- Duplicate values & secondary indexes
  - One option – Dense index for every tuple that exist
  - Buckets
    - Blocks that holds pointers to the same index keys
    - Intermediary level between the index and the tables
- Traditional index
  - Advantage
    - Simple
    - Sequential blocks
  - Disadvantage
    - Not suitable for updates
    - Becomes ugly (loses sequenality and balance) over time

B+ tree

- B+ tree
  - Most popular index structure in RDBMS
  - Advantage

        - Suitable for dynamic updates
        - Balanced
        - Minimum space usage guarantee
    - Disadvantage
        - Non-sequential index blocks
- B+ tree example
    - N pointers, (n-1) keys per node
    - Keys are sorted within a node
    - Balanced: all leaves at same level
- Same non-leaf node (n = 3)
    - At least $\lceil n/2 \rceil$ pointers (except root)
    - At least 2 pointers in the root
- Nodes are never too empty
    - Use at least
        - Non-leaf: $\lceil n/2 \rceil$ pointers
        - Leaf: $\lceil (n-1)/2 \rceil + 1$ pointers
- Insert into B+ tree (simple case)
- Insert into B+ tree (leaf overflow)
    - Split the leaf, insert the first key of the new node
    - Move the second half to a new node
    - Insert the first key of the new node to the parent
- Insert into B+ tree (non-leaf overflow)
    - Find the middle key
    - Move everything on the right to a new node
    - Insert (the middle key, the pointer to the new node) into the parent
- Insert into B+ tree (new root node)
    - Insert (the middle key, the pointer to the new node) into the new root
- Delete from B+ tree (simple case)
    - Underflow (n = 4)
        - Non-leaf < $\lceil n/2 \rceil = 2$ pointers
        - Leaf < $\lceil (n-1)/2 \rceil + 1 = 3$ pointers
- Delete from B+ tree (coalesce with sibling)
    - Move the node across to its sibling if there are rooms available
- Delete from B+ tree (re-distribute)
    - Grab a key from the sibling and move it to the underflowing node
- Delete from B+ tree (coalesce at non-leaf)
    - Push down the parent key into the child node
    - Get the mid-key from parent
    - Push down one of the grand-parent keys into the neighboring parent key
    - Point the child key to the push-down grand-parent key
- Delete from B+ tree (redistribute at non-leaf)
    - Combine the parent and the neighboring keys to make one full node
    - Push down one of the grand-parent keys
    - Push up one of the neighboring parent keys
- B+ tree deletions in practice
    - Coalescing is often <u>not</u> implemented
        - Too hard and not worth it!
- Question on B+ tree
    - SELECT *
      FROM Student
      WHERE sid > 60
    - Very efficient on B+ tree
    - Not efficient with hash tables
- Index creation in SQL
    - CREATE INDEX ON <table> (<attr>, <attr>, …)
    - i.e.,

```
CREATE INDEX ON
Student (sid)
```

        - Creates a B+ tree on the attributes
        - Speeds up lookup on sid
    - Clustering index (in DB2)

```
CREATE INDEX ON
Student (sid) CLUSTER
```

        - Tuples are sequenced by sid

<u>Hash table</u>

- What is a hash table?
    - Hash table
        - Hash function
            - Divide the integer by the key
            - *h*(k): key → integer [0…n]
            - i.e., *h*('Susan') = 7
        - Array for keys: T[0…n]
        - Given a key *k*, store it in T[h(k)]
    - Properties
        - Uniformity – entries are distributed across the table uniformly
        - Randomness – even if two keys are very similar, the hash values will eventually be different
- Why hash table?
    - Direct access
    - saved space – Do not reserve a space for every possible key
- Hashing for DBMS (static hashing)
    - Search key → *h*(key), which points to a (key, record) in disk blocks (buckets)
- Record storage
    - Can store as whole record or store as key and pointer, which points to the record
- Overflow
    - Size of the table is fixed, thus there is always a chance that a bucket would overflow
    - Solutions:
        - Overflow buckets (overflow block chaining) – link to an additional overflow bucket
            - More widely used
        - Open probing – go to the next bucket and look for space
            - Not used very often anymore
- How many empty slots to keep?
    - 50% to 80% of the blocks occupied
        - If less than 50% used
            - Waste space
            - Extra disk look-up time with more blocks that needs to be looked up
        - If more than 80% used
            - Overflow likely to occur
- Major problem of static hashing
    - How to cope with growth?
        - Data tends to grow in size
        - Overflow blocks unavoidable
- Extensible hashing
    - Two ideas
        - Use *i* of *b* bits output by hash function
            - Use the prefix of the first *i* bits of a string of b-bits in length (i.e., use the first 3 bits of a 5-bit hash value)
        - Use directory that maintains pointers to hash buckets (indirection)
            - Maintain a directory and do some indirection
    - Possible problems
        - When there are many duplicates, because there are more copies than digits (?)
            - Still need to use overflow buckets
        - No space occupancy guarantee when values are extremely skewed, thus needing a very good hash function
        - Efficient for equality operator (=), but not efficient for range operators (>, <, etc.)
    - Bucket merge
        - Bucket merge condition
            - Bucket i's are the same
            - First (i-1) bits of the hash key are the same
        - Directory shrink condition
            - All bucket i's are smaller than the directory i
    - Summary
        - Can handle growing files
            - No periodic reorganizations
        - • Indirection
            - Up to 2 disk accesses to access a key
        - • Directory doubles in size
            - Not too bad if the data is not too large