16. Review 2
Main Topics

- Functional Dependencies
- Database design
- Indexing
- Join Algorithms
- Query Processing and Optimization
- Transactions and Concurrency Control
Functional Dependencies

- The functional dependency $X \rightarrow Y$ holds on $R$ if and only if for any legal relations $r(R)$, whenever any two tuples $t_1$ and $t_2$ of $r$ agree on the attributes $X$, they also agree on the attributes $Y$.
- The set of all functional dependencies logically implied by $F$ is the closure of $F$.
- For computing the closure we use Armstrong’s axioms
  - if $Y \subseteq X$, then $X \rightarrow Y$ (reflexivity)
  - if $X \rightarrow Y$, then $ZX \rightarrow ZY$ (augmentation)
  - if $X \rightarrow Y$, and $Y \rightarrow Z$, then $X \rightarrow Z$ (transitivity)

- Given a set of attributes $X$, the closure of $X$ under $F$ (denoted by $X^+$) is the set of attributes that are functionally determined by $X$ under $F$.
- If $X$ determines all attributes, then it is a superkey. If it is also minimal, then it is a candidate key.
- A canonical cover of $F$ is a “minimal” set of functional dependencies equivalent to $F$, with no redundant dependencies or having redundant attributes.
Normalization

- Normalization is the process of decomposing a relation schema $R$ into fragments (i.e., smaller tables) $R_1, R_2, \ldots, R_n$. The goals are:
  - **Lossless decomposition**: The fragments should contain the same information as the original table. Otherwise decomposition results in information loss.
  - **Dependency preservation**: Dependencies should be preserved within each $R_i$, i.e., otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.
  - **Good form**: The fragments $R_i$ should not involve redundancy. A table has redundancy if there is a FD where the LHS is not a key.
Normal Forms

- **BCNF**: for every FD $X \rightarrow Y$, $X$ is a candidate key.
  - There is no redundancy
- **3NF**: a table in BCNF also satisfies 3NF. In addition, 3NF allows FDs where every attribute in $Y$ is prime.
  - There is some redundancy
- **1NF**: every relational table is 1NF because all attribute values are atomic.
BCNF Decomposition Algorithm

Compute $F^+$;
Result = \{R\}; $R_0 = R$; $i = 0$;
While (any $X \rightarrow Y$ in $R_i$ violates BCNF) {
    Create a new table $(X,Y)$;
    Result = (Result-\{R_i\}) U \{(R_i - Y)\} U \{(X,Y)\};
    $R_{i+1} = (R_i - X)$; $i++$;
}

The final tables are in BCNF. The decomposition is lossless-join, does not have redundancy, but may not preserve dependencies.
3NF Decomposition Algorithm

Let $F_c$ be a canonical cover for $F$; $i := 0$;
**for each** $X \rightarrow Y$ in canonical cover $F_c$
    If none of the tables contains $X$ and $Y$
        create $(X,Y)$
    **if** none of the created tables contains a candidate key
        create a table with any candidate key for $R$;

The final tables are in 3NF. The decomposition is both **lossless-join** and **dependency-preserving**, but the tables may have **redundancy**.
Indexing

- Used for faster query processing
- Tree indexes (B+-trees)
  - Good for equality and range queries
- Hash indexes
  - Good only for equality queries
- Multi-dimensional indexes (R-trees, Grid-Files)
  - Good for queries involving two or more attributes
- Specialized indexes (Bitmaps)
Clustering (primary) B+-tree on candidate key

This example corresponds to dense B+-tree index: Every search key value appears in a leaf node.

You may also have sparse B+-tree, e.g., entries in leaf nodes correspond to pages.
Non-clustering (secondary) B+-tree on candidate key

FILE WITH RECORDS

record with search key 1
record with search key 3
record with search key 11

Should be always dense
Join Algorithms - Block Nested Loops

$r \ JOIN \ s$, where $r$ is the **outer relation** and $s$ the **inner relation**

**for each** block $B_r$ of $r$ do begin
  **for each** block $B_s$ of $s$ do begin
    **for each** tuple $t_r$ in $B_r$ do begin
      **for each** tuple $t_s$ in $B_s$ do begin
        if $(t_r, t_s)$ satisfies the join condition
          add $(t_r, t_s)$ to the result

- Use $M$ — 2 disk pages as blocking unit for outer relation, where $M = \text{memory size}$; use remaining 2 pages to buffer inner relation and output
Join Algorithms - Index Nested Loops

• Index lookups can replace file scans if
  – join is an equi-join or natural join and
  – an index is available on the inner relation’s join attribute
  – Can construct an index just to compute a join.

For each tuple $t_r$ in the outer relation $r$, use the index to look up tuples in $s$ that satisfy the join condition with tuple $t_r$. 
Join Algorithms - Merge Join

- Sort both files on the join attribute using **external sorting**
- Scan the two sorted files and produce results for records that match on the join attribute
  - Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Can be used only for equi-joins and natural joins
Join Algorithms - Hash Join

- (Partition step) Partition both files on the same number of buckets using the same hash function on the join attribute.
  - Essentially we create a hash file organization for both files
- (Join step) For each bucket of the small file (build input):
  - Read the entire bucket in memory
  - Read the corresponding bucket of the other file (probe input) page by page and produce results
Query Processing

• **Algorithms for other operations**
  – Duplicate Elimination (Projection) based on Sorting or Hashing
  – Set operations (e.g., Union, Intersection) based on Merge Join or Hash Join

• **Combination of Operations**
  – **Materialization**: store results of an intermediate expression on disk and read them for the next operation
  – **Pipelining**: pass on tuples to subsequent operations even as an operation is being executed (cheaper but not applicable with all algorithms)
Query Optimization

- Goal: to find a good evaluation plan to be executed
  - An evaluation plan is an algebra expression together with the specific algorithm to be used for each operation.

- Query optimizer uses statistics
  - Number of records per table
  - Number of different values per attribute etc.

- Given the statistics, the optimizer estimates the cost of alternative plans and chooses the one with the minimum estimated cost.

- Standard optimization rules:
  - Perform selections before joins
  - Perform small joins before larger ones (for join ordering)
  - Remove useless attributes
Transactions

- **A transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
Concurrency control schemes

- Control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- In addition they need to achieve recoverability
  - if a transaction $T_j$ reads data items previously written by a transaction $T_i$, the commit operation of $T_i$ must appear before the commit operation of $T_j$ (durability property)
- It is also desirable to avoid *cascading rollbacks*
  - when a single transaction failure leads to a series of transaction rollbacks. Solution: only permit reading items written by committed transactions
Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
  - Phase 1: Growing Phase
    - transaction may obtain locks
    - transaction may not release locks
  - Phase 2: Shrinking Phase
    - transaction may release locks
    - transaction may not obtain locks
- The transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).
- May have deadlocks. We can break the deadlocks when they happen, or prevent them by using special variants of 2PL.
Each transaction $T_i$ has a timestamp $TS(T_i)$
- These timestamps determine the serializability order

Each data item $Q$ has two timestamps
- **W-timestamp**$(Q)$ is the largest time-stamp of any transaction that executed **write**(Q) successfully.
- **R-timestamp**$(Q)$ is the largest time-stamp of any transaction that executed **read**(Q) successfully.

Transaction $T_i$ issues a **read**(Q)
- If $TS(T_i) < \text{W-timestamp}(Q)$, then $T_i$ needs to read a value of $Q$ that has been already overwritten.

Suppose a transaction $T_i$ issues a **write**(Q)
- If $TS(T_i) < \text{R-timestamp}(Q)$, then the value of $Q$ that $T_i$ is producing was needed previously
- If $TS(T_i) < \text{W-timestamp}(Q)$, then $T_i$ is attempting to write an obsolete value of $Q$. 
Multiversion Timestamp Protocol

Each data item $Q$ has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Let $Q_k$ be the version of $Q$ whose write timestamp is the largest write timestamp less than or equal to $TS(T_i)$.

1. If transaction $T_i$ issues a read($Q$), then the value returned is the content of version $Q_k$. Reads always succeed.

2. If transaction $T_i$ issues a write($Q$),
   if $TS(T_i) < R$-timestamp($Q_k$), then transaction $T_i$ is rolled back. Some other transaction $T_j$ that (in the serialization order defined by the timestamp values) should read $T_i$'s write, has already read a version created by a transaction older than $T_i$.
   If $TS(T_i) = W$-timestamp($Q_k$), the contents of $Q_k$ are overwritten; $Q_k$ was written before also by $T_i$.
   If $TS(T_i) > W$-timestamp($Q_k$) a new version of $Q$ is created.