Lecture 1: Database Systems – Lecture Notes – Ahmed Nouralla

- **Nature of data**
  - **Information system:** computerized system with components for collecting, storing, processing, and transmitting data.
  - **Database:** collection of related data (recorded facts) representing some aspect of the real world.
  - **Database Management System (DBMS):** computerized system for implementing and manipulating databases.
- **Actors**
  - **Database Administrator (DBA):** the person responsible for monitoring, managing, authorizing access, securing, and optimizing a database.
  - **Database designers:** identifying the data to be stored in a database and choosing the appropriate structure for storing it.
  - **System analysts:** determine the user requirements and specifications for applications.
    - **Application programmers (software engineers)** are the ones to implement, test, debug, and document applications satisfying these requirements.
- **Utilities**
  - **Four basic operations on databases:** Create, Read, Update, Delete (CRUD).
  - **Backup utilities:** to create a backup (dump)
  - **Loading utilities:** to load existing data files into the database.
  - **Reorganization utilities:** to restructure the database system into different file organization.
  - **Performance monitoring:** to provide usage stats for the DBA to improve performance.
- **Relational model**
  - **Concepts**
    - Relational model represents the database as a collection of **tables (relations)**.
    - Each **row (tuple/record)** represents a fact (real-world entity or relation).
    - All values in **columns (attributes)** are of the same datatype (domain of possible values).
    - **Relational schema:** the metadata describing the structure of a relational database model.
  - **Constraints**
    - **Domain constraint:** value of each attribute (A) should belong to the domain of A.
    - **Key constraint:** a key is unique for each tuple.
      - **Key:** an attribute, or a group of attributes that have unique values for each tuple.
        - **Super Key:** any set of attributes that can be used to identify a record.
        - **Candidate Key:** a minimal set that can be used as a super-key.
        - **Primary Key (PK):** chosen to uniquely identify a record in a table (row id).
        - **Composite (compound) Key:** two or more attributes together making a candidate key.
        - **Foreign Key (FK):** used to establish relations between tables.
    - **NULL constraint:** allows/prohibits NULL attribute values.
    - **Entity integrity:** Primary key value cannot be NULL.
    - **Referential integrity constraint:** no null references when referring to tuples in another table (this is to maintain tuple consistency)
- **Functional Dependencies (FD):**
  - Used as a formal tool for analysis of relational schemas.
  - Constraint between two sets of attributes, on the possible tuples that can form a relation state.
  - We say that Y is functionally dependent on X and we write $X \to Y$ if:

$$t1[X] = t2[X] \Rightarrow t1[Y] = t2[Y]$$

    - **Example:** $\mathrm{X} = \{\mathrm{person\_id}\}, \mathrm{Y} = \{\mathrm{person\_name}\}$, i.e., if two persons have the same id, they must also have the same name.
  - **Full (in contrast with partial) FD:** removing any element from X makes the FD not valid.

- **Database normalization:**
  - Takes a relational schema through a series of tests to certify whether it satisfies a certain normal form (1NF, 2NF, 3NF, BCNF, 4NF, 5NF)
  - This is mainly used to minimize redundancy and improve performance.
- **Normal forms:**
  - 1NF ⇐ no multivalued attributes, no composite attributes.
  - 2NF ⇐ 1NF, all non-prime attributes are fully functionally dependent on the relation primary key.
    - A non-prime attribute is not a member of any candidate key.
    - If the PK consists of only one element, the test need not be applied at all.
  - 3NF ⇐ 2NF, no non-prime attribute is transitively dependent on the primary key.
    - $PK \to X \to NP$ shouldn't happen.

Lecture 2:

- **Relational Algebra (RA):**
  - Describes the basic set of (retrieval) operations for the relational model.
  - The result of a relational algebra expression is a relation.
- **RA operations**
  - **From set theory:**
    - UNION, INTERSECTION, SET DIFFERENCE, CARTESIAN (CROSS) PRODUCT.
  - **For relational databases**

	RA	**Example/Equivalent SQL**
SELECT	$\sigma_{Dno=4}(\text{EMPLOYEE})$	SELECT * FROM EMPLOYEE WHERE Dno = 4;
PROJECT	$\Pi_{\text{Lname,Salary}}(\text{EMPLOYEE})$	SELECT Lname, Salary FROM EMPLOYEE;

ASSIGNMENT	$\text{DEB4\_EMPS} \Leftarrow \sigma_{Dno=4}(\text{EMPLOYEE})$	SELECT * AS DEB4_EMPS
		FROM EMPLOYEE WHERE Dno = 4

RENAME	$\rho_{S(B_1,B_2,\ldots,B_n)}(R)$
	• S is the new relation's name, B_i are the new attribute names (assume old ones are A_i).
		ALTER TABLE R RENAME TO S;
		ALTER TABLE R RENAME COLUMN A1 TO B1;
		…
		ALTER TABLE R RENAME COLUMN An TO Bn;

**UNION,
INTERSECTION,
SET DIFFERENCE
CROSS PRODUCT**
	$A \cup B$
	$A \cap B$
	A MINUS B
	$A \times \text{B}$
		SELECT * FROM A UNION ALL SELECT * FROM B;
		SELECT * FROM A INTERSECT ALL SELECT * FROM B;
		SELECT * FROM A EXCEPT SELECT * FROM B;
		SELECT * FROM A CROSS JOIN B

JOIN	$\text{DEPARTMENT} \bowtie_{Mgr_{ssn}=Ssn} \text{EMPLOYEE}$	SELECT * FROM DEPARTMENT INNER JOIN EMPLOYEE ON Mgr_SSn = SSn
Aggregate functions and grouping.	$_{\text{<grouping\_attributes>}}\Im_{\text{<function\_list>}}(R)$	SELECT Count(*) FROM EMPLOYEES

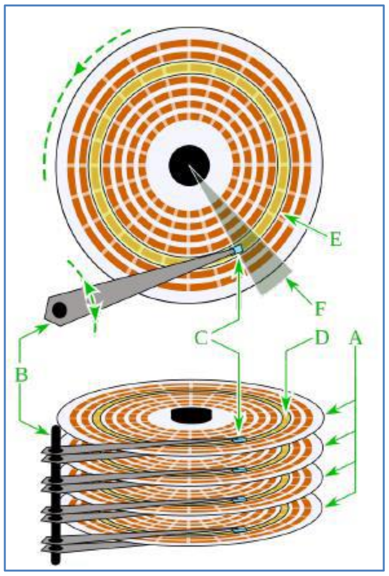
- **More SQL**
  - **(EQUI)JOIN:** joins two tables based on a provided common key.
  - **NATURAL JOIN:** joins two tables based on their common attributes.
  - **EXISTS:** returns Boolean representing whether a query yielded any results.
  - **CASE:** IF/Else logic in databases.
  - **VIEWS:** saving the result of some query for excessive later usage.
  - **TRIGGERS:** action to be taken when some event happens, and some conditions are satisfied.

Lecture 3:

- **Physical storage media for DBs**
    - **Primary storage** (fast access, limited capacity, volatile, more expensive)
        - Main memory, caches, etc.
        - **Example DBMS:** Redis, Aerospike
    - **Secondary storage** (slower access, larger capacity, non-volatile, less expensive, for **online** DBs)
        - **Types:**
            - **Magnetic disks** are the most used.
            - **Flash drives** are also common for medium-sized DBs
            - **SSDs** are used for large DBs for fast retrieval
    - **Tertiary storage:** (slower access, larger capacity, non-volatile, less expensive, for **offline** storage)
        - Cannot be processed by CPU before being copied to the primary storage.
        - **Optical disks (CD-ROMs, DVDs, etc.)**
        - **Tape Libraries**
- **File organization**:
    - The way how data (**DB records**) are stored in a file, how they are stored in disk blocks, and how they are accessed.
        - DB record has a type (numeric, characters, Boolean, date/time)
        - Usually, a file contains records of the same type (that can have fixed/variable length)
        - Binary Large Objects (BLOBs) images/audio/video/etc. are stored separately from their records in a pool of disk blocks.
    - **Primary file organization**
        - **Heap file:** simplest way, not structured, appends new records at the end, often used with **indexes**.
            - **Insertion:** $\boldsymbol{O(1)}$
            - **Searching and deleting:** $\boldsymbol{O(n)}$, deletion introduces holes.
        - **Sorted (sequential) file:** stores records sorted by the value of a particular field (sort key/order field).
            - **Searching is efficient:** $O(log(n))$, especially for range queries.
                - Finding the next record is efficient.
            - **Insertion and deletion** are expensive since order has to be maintained by shifting.
                - Periodic reorganization can be used.
        - **Hashed file:** hashes a particular field (hash key) to determine record placement address.
            - **Insertion, searching, deletion is efficient on average** $\boldsymbol{O(1)}$**,** for equality condition query.
            - **Collisions (multiple objects hashing to the same value)** can be handled by
                - **Open addressing:** check next slots until an empty one is found, deletion is tricky.
                - **Chaining (simplest)**: insert both objects in the same slot (maintain a bucket)
                - **Multiple hashing:** apply a second hash function.

Internal hashing	**External hashing**
hash function maps objects directly to a slot (block address), each slot has only one entry.	hash function maps objects to a bucket number, a bucket may contain more than one entry, collision will happen when bucket is full (less likely)
Extendible hashing *"Hashes are bit strings, lookup uses Tries"*	
▪ Hash function maps objects to a bit string (of 0's and 1's) of length $d' \leq d$, representing a bucket number. ▪ Hash table (directory) contains maximum of 2^d buckets, where d is the global depth of the directory. ▪ Each bucket has a local depth $d' \leq d$ representing the current number of prefix bits this bucket is using. ▪ On collision (bucket is full), the bucket is split and its d' is incremented, if it became bigger than d, then d is also incremented, but the local depths are updated upon need (lazy approach), this will make multiple pointers mapping to the same bucket, but they will also be updated on collision.	

        - **B+ trees:** stores the records in a sophisticated tree-structure for fast access.
        - **Column-based storage**: reduces disk I/O (by compressing similar columnar data)
            - Used for On-Line Analytical Processing (OLAP)
            - **Examples:** Yandex ClickHouse (Column-oriented DBMS), Apache Kudo (Column-oriented data store).
        - **Object-based storage:**
            - Data is managed in a form of uniquely-identified objects rather than files (blocks).
            - Objects contain metadata that can be used to manage them.
            - **Example:** Ceph (software storage platform implementing object storage)
    - **Secondary file organization**
        - An auxiliary access structure allowing faster access to records by "**indexing**" them based on value(s) that are different from the one used in the primary file organization.
- **Access method** defines how files organized in a certain way can be accessed, what operations are possible.
    - The goal of file organization is to avoid linear access (full scan) while accessing records and to locate the record with the minimum number of **block transfers**.
- **Hard drive structure**
    - Hard drive is made of several (packed) identical **discs or platters** (A)
        - Magnetic material protected by plastic or acrylic cover.
    - Each platter has 2 magnetic **read/write heads** (for each side) (C)
        - #Platters per disk = 2 * #Heads per disk
        - Sometimes, only one face (1 head) is used.
    - An **actuator** (B) with an **arm** is used to group read/write heads
    - Each platter has several **tracks** (E)
    - Each track has several **sectors** (F)
        - A block represents the logically addressable minimum unit of data
        - Block can be of the same/different size than a sector.
        - Each sector has a capacity (in bytes)
    - **Cylinder**: a stack of tracks
        - #Tracks per platter = #Cylinders per disk
        - #Tracks per cylinder = #Platters per disk
    - Data stored on one cylinder can be retrieved much faster than if it was stored on different cylinders.



- **Addressing data:** Cylinder no. → track no. → block no.
- **Accessing data**
    - **Rotational delay (latency):** time needed for the disk to rotate for the arm to be in correct position.
    - **Seek time:** time needed for the head to move to the right track
    - **Block transfer time (smallest):** time to transfer the data.
- **Minimizing access time**
    - **Buffering data in memory:**
        - **Buffer pool:** the part in main memory where data coming from disk is buffered before usage
            - Size is a parameter for the DBMS controlled by DBA
        - **Buffer manager:** part of the DBMS that responds to data requests and decides the buffering mechanism.
            - It works exactly like a cache for disk blocks with the goal to maximize hit rate.
            - Pin-count (indicating block popularity) and dirty (modified) bit are used.
        - **Buffer (page) replacement algorithms**
            - **Least Recently Used (LRU):** removes the LRU page
            - **First-In-First-Out (FIFO):** removes the oldest page, adds the new one at tail.
            - **Clock algorithm:** uses circular buffer with R bit (recently used) for each block.
                - The clock hand resets $R = 0$ (if it was 1) for pages it visits and advances.
                - The page with $R = 0$ is the one to be evicted.

- **Proper file organization:** keep related data in contagious blocks
    - **Unspanned records**: a fixed number of records is stored in each block, consecutive access.
    - **Spanned records**: make use of wasted space for storing parts of records with a pointer at the end of a block pointing to the block containing the rest of the record.

> - Assume block size = B bytes, file has n fixed-length records, each of size = R bytes
> - Blocking factor of a file (number of records per block) $bfr = \lfloor B/R \rfloor$
> - Wasted space in each block = $B - bfr * R$
> - File will occupy $\left\lceil \frac{n}{bfr} \right\rceil$ blocks = $B * \left\lceil \frac{n}{bfr} \right\rceil \geq$ nR bytes

- **Reading data ahead of request** to minimize seek time.
- **Proper scheduling of I/O requests** (e.g., elevator algorithm)
- **Temporary hold writes** (using log disks to group write operations) to minimize arm movement.
- **Use of SSD or flash memory for recovery.**

Lecture 4:

- **Indexes** [1], [2]:
    - Additional access structures (files on disk) used to speed up responses to certain types of queries by providing secondary (faster) access paths to the DB records (instead of sequential scan).

"You query a huge database for a record or a range of records. Looking through all records is slow, so you maintain ***an index*** *on a database column(s), allowing you to use the search value itself (or a function on it) to know where to start looking for resulting records, (i.e., minimizing the search space)"*

Sparse indexes	Dense indexes
There is an index file entry for every block in the data file (one record from each block is included in the index file).	There is an index file entry for every record in the DB. Takes more space, but faster in performance.

- **Index file structure:** a table with two columns: Search Key, Data Reference.

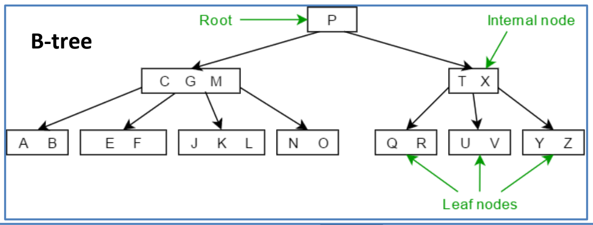
"Key" ⟹ column values are unique
"Field" ⟹ column values are not necessarily unique

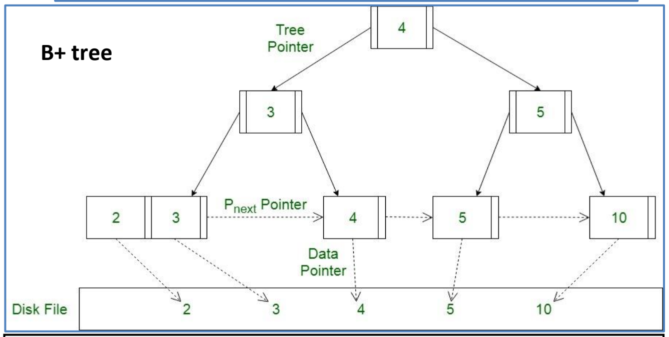
- **SK:** the first column in an index file containing some/all of the column values (or a function on them)
    - Names for the SK.
        - **Index field/key:** when the values are not copied from some database column, but rather a function on them.
        - **Ordering field/key:** if the index is ordered, search key column can be called an ordering field.
        - **Clustering field** (in clustering indexes): index is ordered, search keys are if it's not necessarily unique, but used to order records in non-increasing/non-decreasing order.
    - **Ordered index:** an index where data in SK are sorted
        - Ordered indexes utilize binary search, which improves performance.
- **DR:** a pointer/set of pointers to the datafile
    - DR may point to:
        - A datafile block(s) (primary indexes)
        - The database record itself (secondary/dense indexes)
        - Or another (smaller) index file to check there (multilevel indexes).
    - **Single-level index:** the pointers point directly to data and not to a bucket of pointers
- **Index types** (many possible classifications exist):
    - **Single-level ordered indexes**
        - **Primary index (sparse, automatically created):**
            - **Ordering key:** the primary key (an attribute that has to have **unique values**).
            - **Data reference:** a pointer to the disk block that contains the record with that ordering key.
                - Number of index entries = Number of disk blocks in the data file
                - One entry (Key, Pointer) corresponds to one block.
            - **Problem:** insertion and deletion will require updating indexes.
            - **Solution:**
                - Use a (sorted) linked list containing extra records that exceed block size (for insertions).
                - Use deletion markers to handle deletions.
        - **Clustering index (sparse):**
            - **Ordering (clustering) field:** **doesn't have to** have unique values.
            - **Data reference:** points to the disk block that has the first occurrence of the clustering field.
            - **To solve the problem with insertion/deletion**, we can reserve a block/cluster for each possible value of the clustering field.
        - **Secondary (non-clustering) index (dense, can be single/multi-level):**
            - **Indexing field:** an attribute that is not necessarily ordering/unique,
            - **Data reference:** points to the record associated with that indexing field.
                - In case of multilevel, the pointer points to a subset index file.

- **Type 1**: uses a unique unordered key (i.e., a candidate key)
  - Needs more storage space and longer search time than a primary index.
- **Type 2:** uses a non-unique unordered field, can be implemented by:
  - Including duplicate index entries with the same indexing field but different pointers.
  - Or using an extra level of indirection (pointer to another index file, described above).

- **Multi-level indexes**
  - Reduces the number of disk block/record accesses while doing binary search on an ordered index file, by introducing multiple levels of index files, organized as an ordered tree.
  - Dynamic multi-level indexes deal with insertion/deletion problems by using appropriate algorithms and data structures (e.g., B-trees, B+ trees) for creating/deleting blocks when the data grows/shrinks.

B-trees:
- Balanced search trees (all leaves have the same depth) that provide fast, direct access to data in disk by minimizing disk I/O using the idea of branching (only a subset of data is examined during search).
- Every node contains:
  - Keys (in non-decreasing order)
  - Key count.
  - Boolean IsLeaf.
- (Minimum) degree (t) = lower bound on the number of children a node can have
- Maximum degree (m) = order = upper bound on the number of children a node can have
- Branching factor = average number of children per node
  - $m = 2t,\ t = \lceil m/2 \rceil$
  - Tree height $h \leq \log_t \frac{n+1}{2}, n = \#\text{Keys}$
- A node containing n keys, has $n + 1$ children.
- B-TREE-SEARCH is $O(\log_t(n))$
- B-TREE-INSERT and B-TREE-DELETE are $O(t * \log_t(n))$





B+ trees differ from B-trees by having the data pointers only at leaf nodes, which makes search faster and insertion/deletion easier (more entries can be packed into an internal node, as it doesn't have data pointers)
B+ trees are the common/default structure used to generate on-demand indexes.

- **Hash indexes:** the index file is a hash file (divided into numbered buckets)
  - **Ordering field:** data of the column being indexed, ordered, and divided into numbered buckets.
    - The bucket number is the result of hashing the search value.
  - **Pointer:** points to a block/record.
- **Bitmap indexes:**
  - **Ordering field:** a bit string for each unique value $V_{i \in [1..m]}$ in the column C being indexed.
    - The bit string has a length n (the number of records) in the DB
    - A value of '1' means that $C[i] = V_i$ and '0' otherwise.
    - AND/OR queries are done efficiently by doing bitwise operations on fields of different bitmap indexes.
  - **Pointer:** points to a block/record.
  - Efficient for queries on multiple keys (bitwise operations are used), with many rows (takes small space), but small number of unique elements (since an entry for each unique value is maintained in the index file)
    - **A typical example:** the gender field in a table.
  - **Problem:** on record **deletion**, renumbering rows and shifting bits becomes expensive.
  - **Solution:** another "existence" bitmap can be used (0 for zombie rows, 1 for rows that exist).
  - The problem with insertion (an entry has to be added to all bitmap indexes) still constitutes an overhead
    - Adding new rows in the place of deleted rows will slightly reduce that overhead.

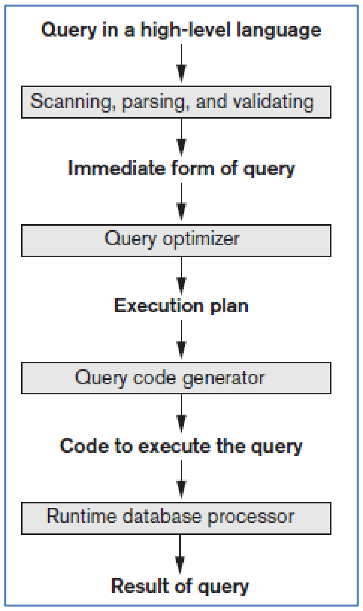
- **Indexes on multiple keys:**
    - Useful when a certain combination of attributes is frequently used together in queries.
    - **Implementations**
        - **Ordered index:**
            - **Ordering field:** the tuple of values of the columns, ordered lexicographically in index file.
            - **Pointer:** points to the record with that field.
        - **Partitioned hashing** (suitable only for equality (not range) queries)
            - **Ordering field:** the tuple of values of the columns, divided into ordered buckets
                - The bucket number can be calculated given the tuple of search values v_i.
                - Bucket number = $concatenate\left(hash(v_1), \dots, hash(v_n)\right)$
            - **Pointer:** points to the record with that search value.

In PostgreSQL, Only B-tree, GiST, GIN, and BRIN index types can be used with multiple (up to 32) columns.

- **Issues concerning indexes**
    - **Index creation:**
        - **Bulk loading the index:** insertion of a large number of entries into the index, it's a linear (costly) operation that is done once upon creating the index.
        - **Primary/Clustering indexes** are the most expensive type of indexes since they require the data to be physically ordered in disk.
    - **Tuning indexes**
        - Some indexes may be needed and not present, others may be redundant.
        - Some indexes may introduce high overhead due to frequent updates on their columns.
- **Factors that influence physical database design.**
    - **Database queries and transactions**
        - What is the use of the database?
        - What type of queries are generally required?
    - **Expected frequency of invocation of queries and transactions**
        - **80-20 rule:** approx. 20% of the queries (the vital ones) are responsible for 80% of the processing.
    - **Time constraints of queries and transactions**
        - The attributes accessed by time-constrained queries are the ones that should be considered for primary indexes (since they are the fastest).
    - **Expected frequencies of update operations**
        - Updating indexes can slow down insert operations.
    - **Uniqueness constraints on attributes**
        - Access paths should be specified on all candidate keys.

Database transaction: represents any change in the database.

Lecture 5:


Query in a high-level language
Scanning, parsing, and validating
Immediate form of query
Query optimizer
Execution plan
Query code generator
Code to execute the query
Runtime database processor
Result of query

- **Query processing (done by the DBMS) steps:**
  - **Scan:** identify query tokens (keywords, attribute/relation names, etc.)
  - **Parse:** check syntax against language grammar, does it follow the rules?
  - **Validate:** check semantics (validity) of attribute/relation names
  - **Construct internal representation (query tree):** helps the DBMS decide the order of operations during query execution.
    - **Query tree can be represented as a tree with**
      - **Internal nodes**: "extended" relational algebra operators + operands (RA expression)
        - Extended RA includes aggregate operators (MIN, MAX, SUM, COUNT, etc.)
      - **Leaf nodes:** tables being queried, query execution starts from bottom to top.
    - **Query block:**
      - Basic unit that can be translated into RA expression.
      - A single SELECT – FROM – WHERE – [HAVING – GROUP BY] clause.
  - **Design query plan:** the DBMS provides an optimized execution strategy (steps).
    - Query optimizer choose a query plan for each query block
      - Especially the **correlated** complex nested subqueries.

Correlated nested queries:
An inner query that uses a variable from the outer query. A smart DBMS will cache the result of the inner query to avoid computing it multiple times.

- **External sorting:** class of sorting algorithms that operate on massive amount of data (large record files) that doesn't fit in main memory.
  - **Sort-merge:**
    - An algorithm for external sorting that works as follows:
      - Sort **runs** (small subfiles of the main file), i.e., copy them into main memory, then use an internal sorting algorithm to sort them.
      - Merge the results and write them back to the disk.
    - It requires buffer space in main memory for actual sorting/merging.
    - Number of runs $= \text{ceil}\left(\frac{\text{number of disk blocks occupied by file}}{\text{number of available main memory buffers}}\right) \Leftrightarrow n_R = \left\lceil \frac{b}{n_B} \right\rceil$
    - Worst case $O(n * log(n))$

Sorting in DB is used when
1. ORDER BY clause is specified.
2. DISTINCT data are required.
3. With JOIN/INTERSECTION/UNION that uses sort-merge or similar algorithms

- **Searching:**
  - **File scan:** searching for records in a file.
    - **Linear search:** $O(n)$
    - **Binary search:** $O(log(n))$
  - **Index scan:** searching in an indexed column, uses the algorithms above depending on the type of index.
    - **Primary index:** equality/inequality conditions.
    - **Clustering index:** equality condition, retrieves 0 or more columns.
    - **Hash index:** equality condition, retrieves one record at most.
    - **Btree index:** equality/inequality conditions.
    - **Bitmap index:** equality on a set of values
      - This is done by OR-ing the bitmap for each element in the set.
    - **Functional index:** matching is based on a function on the functionally-indexed attribute.

        - **Conjunctive selection (AND) algorithms**
            - Retrieve all records satisfying the first condition, then continue checking other conditions and exclude those which fail.
            - If a composite index exists on the combined fields, we use it directly.
            - If secondary indexes exist on some fields, use them to get all record pointers and intersect the results, rest of the conditions are handled using the first method.
        - **Disjunctive selection (OR) algorithms** (much harder to process and optimize)
            - Result is the union of results for each condition, algorithm is decided by the query optimizer.
    - **DB Metadata:** data about the database are stored in the DBMS and are used to estimate the percentage of records that will be retrieved by queries.
        - r_R = number of records in relation R
        - R = width of relation (number of bytes)
        - b = number of disk blocks occupied by relation
        - bfr = number of rows per disk block
        - $\max(A, R)$, $\min(A, R)$ = max, min value of attribute A in relation R

DB catalog: contains DB instance metadata (tables, views, indexes, users, etc.)

- **Join (time-consuming operation)**
    - **Two-way joins:** joins on two files.
    - **Implementations** $R \bowtie_{\text{R.A = S.B}} S$
        - **Nested loop/block join (default, $O(n^2)$):**
            - Retrieve all possible combinations and exclude those who don't satisfy join condition.
            - It's better to use the file with fewer blocks as the outer loop since it minimizes block accesses.
        - **Index-based join** (assuming column B is indexed):
            - Retrieve all records in R and use index on B to retrieve only the records from S that satisfy join condition. `for t in R: retrieve_all(S[B]=t[A])`
        - **Sort-merge join** (assuming records of R and S are ordered by the value of A or B respectively)
            - One linear pass will be enough since $R[A] = S[B]$ will only happen if A, B have the same block order in their list.

n = Number of memory buffers/blocks available.
- One buffer has one block from inner loop file.
- Another buffer is used for storing results, if it's full, contents are moved to the result disk file.
A join on tables **O**uter (b_o $blocks$) and **I**nner (b_i $blocks$) requires $b_o + \left\lceil \frac{b_o}{n-2} \right\rceil * b_i$ block accesses.

- **Set operations:**
    - **Cartesian product:** computationally expensive
    - **Union, Intersection, Set-difference implementations:**
        - Can be done in one pass after sorting both (type compatible) tables.
        - Hash the results of one query, then compare all results of the second query with the hashed ones.
- **Aggregate operations:**
    - **MIN, MAX, SUM, AVERAGE, COUNT:** using a linear pass or an index.
        - MIN/MAX for a b-tree indexed attribute is done easily, leftmost element is MIN, rightmost element is max, they may also be stored in DB catalog.
        - COUNT is usually stored as metadata.
        - SUM, AVERAGE can be calculated only from a dense index that has an entry for each record.
    - **GROUP BY**
        - Sorting/Hashing the grouping attributes to partition the file into groups.
        - A clustering index already does the job since the equal values will be consecutive in the index.

GROUP BY: merges all the results sharing the same values of a particular attribute A, it's mostly used with an aggregate function on some value containing info about A that needs to be collected. **For example:** getting a table of total payments sum, grouped by the client's name.

- **Pipelining queries:** a query result can be directly redirected as an input for the next query (a query plan has to be prepared), this improves performance since no temporary tables are stored in disk ⇒ no materialized evaluation.

Lecture 6:

- **Query optimization (done by the DBMS query optimizer):**
  - Selecting the best possible strategy for query execution using the available information about the schema, the contents of relations involved, whether there are nested correlated queries, and whether there are indexes to be utilized.
  - **Heuristics** rules are used to modify the internal representation (query tree) before execution.
    - **Example:** apply SELECT/PROJECT before JOIN, since the first ones reduce the file size and the second one is likely to expand it.
  - **Estimates and cost-comparisons** are also used to provide DB-specific optimization strategies
    - **Cost-based query optimization:** is the optimization strategy that tries to **minimize** a cost function (of some factors and their significance) by comparing possible execution strategies and choosing the best.
    - **Examples of such factors:**
      - Access (I/O) to secondary storage.
      - Disk storage (cost of storing intermediate results)
      - Computation (CPU) cost (for in-memory operations)
      - Memory usage (the number(size) of buffers utilized)
      - Communication cost (of transferring tables among various computers)
    - **Catalog information used in cost functions:**
      - Size of each file (containing records of the same type)
      - Primary file organization of each file
      - Indexes and their attributes, and the number of levels in multilevel indexes
      - Number of distinct values $NDV(A, R)$, **attribute selectivity**, number of blocks taken by indexes.
        - **Attribute selectivity:** percentage of records satisfying an equality condition on some attribute.
        - **Selection cardinality** = attribute selectivity $* \ r$
      - A histogram (distribution of column values in DB tables)
    - **Cost function for SELECT operations:**
      - **Linear search:**
        - For inequality condition: $C = b$
        - For equality condition $C = \frac{b}{2}$, or b if value does not exist.
      - **Binary search:** $C = log_2(b) + \left\lceil \frac{s}{bfr} \right\rceil - 1$
      - **Primary index search:** $C =$ number of index levels $+ 1$
      - **Hash index search:** $C = 1$ for static/linear hashing, $C = 2$ for extendible hashing.
      - **Secondary (B+ tree) index search:**
        - Equality condition of key attribute: $C = \mathrm{x} + 1$
        - Equality condition of non-key attribute: $C = x + 1 + s$
        - Inequality condition (range query): $C = x + \frac{b}{2} + \frac{r}{2}$

$x =$ number of index levels
$s(A) =$ selection cardinality.
$b(A) =$ number of first-level index blocks.
$r(X) =$ number of records in relation X

- **General transformation rules for RA operations** (using them wisely will improve performance)
  - **σ,×,⋈ are commutative:**
    - $\sigma_{c_1}\left(\sigma_{c_2}(R)\right) \equiv \sigma_{c_2}\left(\sigma_{c_1}(R)\right),\ R \times S = S \times R,\ R \bowtie_{c} \mathrm{S} = S \bowtie_{c} \mathrm{R}$
  - **Cascade of σ:** always do it as it allows moving selects down the tree.
    - $\sigma_{c_1\ AND\ \ldots AND\ c_n} \equiv \sigma_{c_1}\left(\ldots\left(\sigma_{c_n}(R)\right)\ldots\right)$

General tree optimization strategies:

- Move the most restrictive σ/π (the one that selects the least number of records, or has the smallest size) to the leaf (to be executed first)

- Cascade π whenever the results of some column are not needed for next operation or final result.

- Try to avoid cartesian product by reordering leaves, or using the underlined transformation.

- **Cascade of π:**
  - $\pi_{list_1}\left(\ldots\left(\pi_{list_n}(R)\right)\ldots\right) \equiv \pi_{list_1}(R)$ "all but the last one can be ignored."
- **Commuting σ with π (only if selection condition involves the same attributes to be projected)**
  - $\pi_{A_1,\ldots A_n}\left(\sigma_{condition}(R)\right) = \sigma_{condition}\left(\pi_{A_1,\ldots,A_n}(R)\right)$
- **Commuting σ with ⋈ (Or ×)**
  - If all the selection conditions involve rows only from one of the tables (e.g., R), the SELECT can be applied on one of them before the JOIN to improve performance: $\sigma_c(R \bowtie S) = \sigma_c(R) \bowtie S$
  - If the selection condition $c = c_1\ AND\ c_2$ where c_1 involves R only and c_2 involves S only then $\sigma_c(R \bowtie S) = \sigma_{c_1}(R) \bowtie \sigma_{c_2}(S)$
- **De Morgan's laws can also be used to optimize conditions.**
  - NOT (c_1 AND c_2) ≡ (NOT c_1) OR (NOTc_2)
  - NOT (c_1 OR c_2) ≡ (NOT c_1) AND (NOTc_2)
- Commutativity of ∪ and ∩
- Associativity of ×, ⋈, ∪, ∩
- Commuting σ with set operations. $\sigma_c(R\ \theta\ S) \equiv \left(\sigma_c(R)\right)\theta\left(\sigma_c(S)\right),\ \theta = [\cup \mid \cap \mid -]$
- The π operation commutes with ∪: $\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$
- Converting a (σ, ×) sequence into ⋈ : $(\sigma_c(R \times S)) \equiv (R_c \bowtie_c S)$
- Pushing σ in conjunction with set difference $\sigma_c(R - S) = \sigma_c(R) - \sigma_c(S)$
- Pushing σ to only one argument in ∩: $\sigma_c(R \cap S) = \sigma_c(R) \cap S$
- Trivial transformations: $S = \emptyset \Longrightarrow R \cup S = R, c = true \Longrightarrow \sigma_c(R) = R$

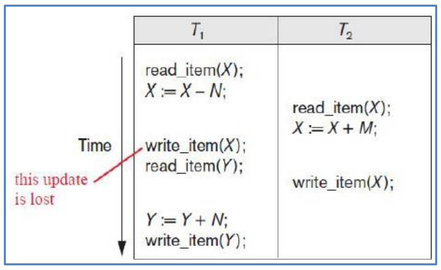
Updating a materialized view:

- **Immediate update:** whenever any relation is updated.
- **Lazy update:** update upon demand.
- **Periodic (deferred) update:** refreshes the view on certain intervals.

- **Query plan (access methods + algorithms to be used) optimizations.**
  - Try to pipeline queries whenever possible.
  - Utilize indexes.
  - Try to unnest/decorrelate nested correlated queries into JOINs.
  - Convert queries in FROM clauses to **inline views** (temporary tables) and append them before selecting.
    - This may be invalid if the view is more complex (DISTINCT, OUTER JOIN, AGGREGATION, GROUP BY, SET OPERATIONS, etc.)

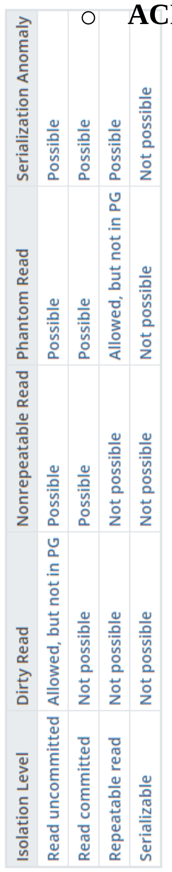
Lecture 7 (week8)

- **Multiuser DBMS**
    - Multiple users can access the database resources (**data items**) concurrently.
        - **Data item:** a database record, disk block, or individual field (attribute value for some record).
        - **Granularity:** size of the data item (what portion of DB it represents).
    - **Interleaved concurrency:** the CPU can be switched to run other process rather that remaining idle during I/O.
    - **Parallelism:** running multiple processes at the same time, requires multiple CPUs.
- **Transactions:**
    - The logical unit of database processing (set of database **access operations**) that must be completed entirely (committed), or otherwise uncommitted entirely (rolled back with no effects) to ensure correctness.
        - Transactions are necessary in large databases with many concurrent operations executed by different users at the same time.
    - **Transaction (access) operations:**
        - A transaction reads/writes data items, concepts apply to any data item.
        - Transactions can be embedded within a program (e.g., a python script), or declared in SQL.
        - Transactions in SQL are written inside a BEGIN-END clause.
            - **Typical SQL transaction operations:** insert, read, update, delete.
            - **Read-only** transaction only retrieves data items.
            - **Read-write** transaction can retrieve/modify data items.
        - **Read operation** involves finding the location of the item to be read, copying it to a buffer, then storing it to the program variable X.
            - We say that $X \in$ read-set of the transaction.
        - **Write operation** involves finding the location of the disk block to be written to, copying that disk block to a buffer, copy the contents program variable X into that buffer, updating the disk block with the new data from the buffer (can be postponed).
            - We say that $X \in$ write-set of the transaction.
    - **Problems caused by interleaved concurrent execution of transactions (T1 and T2)**
        - **Lost update problem:**
            - T1 reads then updates X and didn't commit changes yet.
            - T2 reads (old, un-updated) X, then updates the old copy.
            - T1 commits then T2 commits.
            - The committed update by T1 is **lost,** since T2 has modified an old copy.



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

        - **Temporary update (dirty read) problem:**
            - T1 reads then updates X
            - T2 reads the updated X
            - T1 rolls back (fails), now T2 has an invalid (temporary) value for X
        - **Incorrect summary problem**
            - T1 is doing an aggregate operation on some data items.
            - T2 is updating one or more of the data items.
            - Aggregation result will be wrong as some values will be outdated.
        - **Unrepeatable read problem**
            - T1 reads X, T2 updates X, T1 reads X again, wtf, this read is unrepeatable.
        - **Phantom read problem [check]**
            - T1 runs Q, T2 inserts/updates some DB records, T1 runs Q again, different results got.
                - Different results = different rows are being selected; no row content is being changed.
        - **Serialization anomaly**
            - Committing T1 before T2 brings different results than if T2 was executed first.



Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

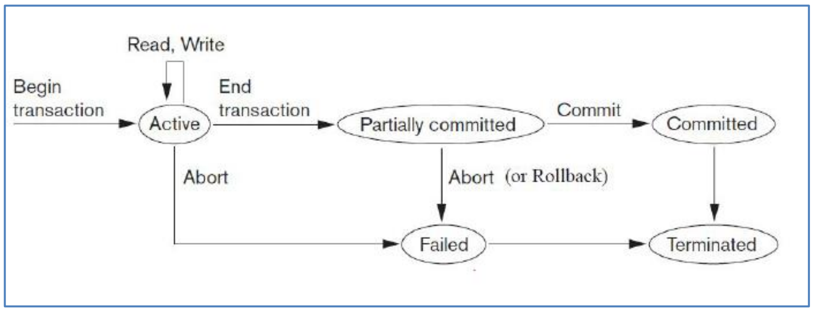
- **ACID properties:** transaction operations must be ACID (Atomic, Consistent, Isolated, Durable)
  - **Atomic:** should be executed as a whole or not executed at all ♫
  - **Consistent:** a single non-interfered transaction should take the DB from a consistent state to another (rollback on any DB rule violation)
    - Consistent state ⇔ maintains data integrity **(check lecture 1)**
  - **Isolated:** two transactions executing concurrently shouldn't interfere with each other.
    - **Isolation levels**:
      - **0. Read uncommitted (dirty read):**
        - Allows a transaction to read uncommitted changes (done by other concurrent transactions) to the data, which may result in temporary update problem.
      - **1. Read committed (default in PostgreSQL)**
        - Transactions can only read committed data, preventing dirty reads.
      - **2. Repeatable read**
        - A transaction can only read data committed before it began executing.
        - No lost updates, no dirty reads.
      - **3. Serializable (true isolation)**
        - Emulates serial execution of transactions.
        - No lost updates, no dirty reads, ~~no un~~repeatable reads.
    - **Snapshot isolation**
  - **Durable:** committed changes should persist, not lost because of any failure.
    - Achieved through changelogs and log buffers.

Lecture 9

*"To enforce ACID properties on transactions, **concurrency control** and **recovery methods** are done by the DBMS."*

- **Transaction failure and recovery:**
  - **Failure scenarios**
    - **System crash:** (hardware, software, network error) during the transaction
    - **Transaction runtime error** (e.g., integer overflow, division by zero, etc.)
    - **Local errors, exceptions** (e.g., data item not found)
    - **Concurrency control** enforces transaction failure (e.g., to resolve a deadlock)
    - **Disk failure, physical catastrophe** (less common)
  - **Failure recovery**
    - The system keeps track of each transaction **state** (started, executing, ended, committed, or aborted)
    - The system maintains a sequential, append-only log on disk with all transaction operations, to be used for "**transaction log recovery**".
      - Log is only affected by disk/catastrophic failures.
      - Log entries are stored in main memory **log buffers** before being written to disk.
      - **Write-Ahead Log (WAL)** is used by some DBMSs, ensures data integrity by recording every change in datafiles before committing these changes (force-writing).

Transaction log entries
[start_transaction, T]
[write_item, T, X, old_value, new_value]
[read_item, T, X]
[commit, T]
[abort, T]

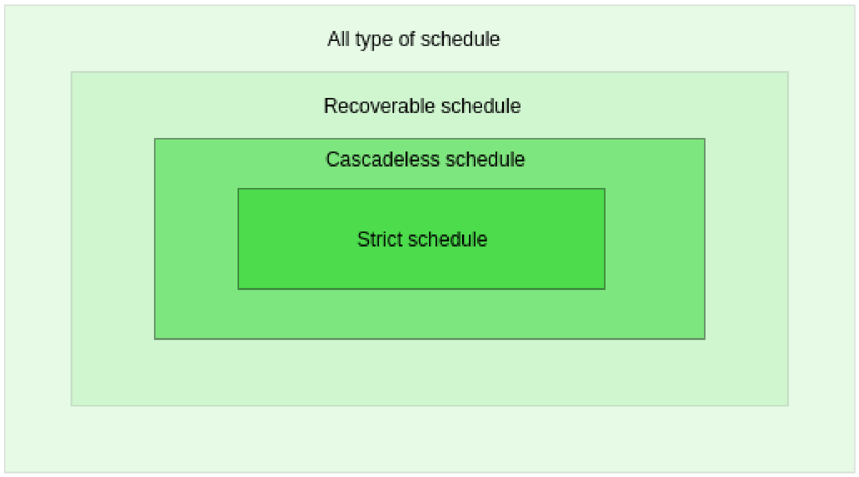


Transaction state-transition diagram

Lecture 8 (week9)

- **Conflicts**
  - **Conflicting operations:** two read/write operations are said to conflict if all of the following is true:
    - They belong to different transactions.
    - They access (read/write) the same item X.
    - At least one of them is writing to X.
  - **Conflict types:**
    - **Read-write conflict:** $r1(X), w2(X)$ results in a different outcome than $w2(X), r1(X)$
    - **Write-write conflict:** $w1(X), w2(X)$ results in a different outcome than $w2(X), w1(X)$
- **Schedule (history)**
  - Order of concurrent transaction execution decided by the DBMS**.**
  - A good choice of schedule facilitates recovery, a bad choice can make recovery from a failing transaction (during schedule execution) impossible.
  - **Recoverability**
    - **Recoverable schedule:** a schedule that makes it possible to rollback a failing transaction safely (completely wiping effects of failing transaction, without affecting the data committed by other transactions)
      - **Example:** $r1(X);\ r2(X);\ w1(X);\ w2(X);\ c2;\ c1;$
      - A recoverable schedule can be **cascading** or **cascade-less**.
      - A cascade-less schedule can be **strict** or non-strict.
    - **Non-recoverable schedule:** a schedule having some transaction that read temporary data and is committed (it can't be rolled back due to durability property of transactions)
      - **Temporary data:** was modified by a transaction that failed later.

- **Cascading schedule:** requires rolling back an uncommitted transaction when it fails (time-consuming)
- **Cascade-less schedule:** ensures that a transaction should commit only after the data it read is committed, thus won't require rolling back.
- **Strict schedule:** ensures that transactions don't access data until the last transaction that modified it is committed.



Lecture 9 (week10)

- **Lock**
    - A variable associated with a data item X in the DB describing its status (possible operations on X) and is used to manage access to X by concurrent operations.
    - **Lock conversion**
        - Upgrading lock: acquiring a more restrictive lock on a locked item.
        - Downgrading lock: acquiring a less restrictive lock (or unlocking) a locked item.
    - **Lock types**

Q: What data item to lock? record, field, file, disk block, or entire database.
A: Each choice affects performance of concurrency control and recovery in different way, the best granularity to choose depends on the given transaction.

Multiple granularity locking: hierarchically breaking up the database into a tree of blocks (files, pages, records) with the whole DB at the root, each block (and all its children) can then be locked as a unit and accessed individually.

	Binary lock	**Shared/Exclusive (Read/Write) lock**
States	lock(X) = 0 • data item cannot be accessed at all. lock(X) = 1 • data item can be accessed for read/write.	lock(X) = read_locked • a transaction is reading X now, other transactions can still read X (shared access), but none should write to it. lock(X) = write_locked • a transaction writing to X now (exclusive access), no other transactions should read/write (to) X.
Operations	lock(X), unlock(X)	read_lock(X), write_lock(X), unlock(X)
Comments	- Simple - Too restrictive (at most one transaction has access to the data at a time) - Not widely used. - DBMS lock manager subsystem control access to locks and maintains a hash file containing locked items.	- More complex - Less restrictive (multiple transactions can read X at the same time if it's not write_locked) - Used in DB locking schemes as it provides more general locking capabilities.

- **Transaction deadlock**
    - Each transaction in the set is waiting for another transaction in the set to release a lock.
    - **Ex:** T1 is waiting for T2 to release the lock on X, while T2 is waiting for T1 to release the lock on Y.
    - **Prevention strategies**
        - **Lock all** the items a transaction needs **in advance** or wait then try again **(not practical)**
        - **Wait-die**
            - **O** trying to acquire a resource held by **Y** → **wait** (until the resource is released by Y).
            - **Y** trying to acquire a resource held by **O** → **die** (abort and restart later).
        - **Wound-wait**
            - **O** trying to acquire a resource held by **Y** → **wound** (make **Y** aborts and restart later).
            - **Y** trying to acquire a resource held by **O** → **wait** (until the resource is released by O).
        - **Detect** the deadlock by constructing the resource graph **and abort** a random victim (better choose a younger one that made less updates).
        - If a transaction **timeout**, assume a deadlock, abort and restart later (practical solution: simple, low overhead).

Transaction timestamp TS(T)
unique id for each transaction representing its starting time.

Older transactions have smaller TS.
Younger transactions have higher TS.

- It can be proven that both **wait-die** and **wound-wait** are deadlock free and prevent starvation (transaction waiting forever to acquire lock although there is no deadlock).

- These methods however may cause unnecessary aborts/restarts.

- **Concurrency control:** a mechanism that guarantees consistency and isolation of concurrent transactions, is affected by the granularity of data items.
  - **Two-Phase Locking (2PL)**
    - Locking protocol in which all locking operations (or locking upgrades) at the **first (expanding/growing) phase** precedes all unlocking operations (or locking downgrades) at the **second (shrinking) phase** in the transaction schedule.
    - If every transaction follows the 2PL protocol, the schedule is guaranteed to be serializable (true isolation).
    - This approach however limits the amount of concurrency, since it doesn't consider the access order done by transactions (a transaction writing to X will lock X during its entire execution time).
  - **Timestamp-Ordering algorithm:**
    - Concurrency control approach that uses timestamps to order transaction execution to enforce a serial-equivalent order; deadlocks can't occur, **but starvation may occur (due to cyclic restart)**.
    - Interleaving is still possible, as long as timestamp order is respected for each pair of conflicting operations.
    - **Algorithm:**

```
if T.write_item(X):
    if read_TS(X) > TS(T) or write_TS(X) > TS(X):
        reject(T) # reject operation: abort and rollback T
    else:
        accept(T) # accept operation
        write_TS(X) = TS(T)

if T.read_item(X):
    if write_TS(X) > TS(T):
        reject(T) # reject operation: abort and rollback T
    else:
        accept(T) # accept operation
        read_TS(X) = max(read_TS(X), TS(X))
```

read_TS(X): timestamp of the last transaction that read X successfully.

write_TS(X): timestamp of the last transaction that wrote to X successfully.

  - **Multi-Version Concurrency Control (MVCC)**
    - Widely used in relational and NoSQL databases, implementations vary.
    - Keep copies of old values for data items (several versions) when they are updated by transactions.
    - When a transaction request to read item, the appropriate version of the item is being used, this maintains serializability of the schedule.
      - The value being retrieved depends on the transaction performing the read.
    - **Temporal DB:** keeps track of all (timestamped) changes on the DB.
  - **Optimistic (validation-based) Concurrency Control (OCC)**
    - Requires no overhead checks during transaction execution (before data item access).
    - Useful when transactions usually complete without conflicts.
    - **Algorithm (3 phases: read, validate, write)**
      - Assume no conflicts will happen (optimistic), perform all transaction operations.
        - **Read** data committed by other transactions.
        - Updates only to local copies for possible rollback.
      - Before committing T, **validate** whether the data it read was modified by other transaction.
        - If no, commit (**write** local copies to DB) safely.
        - If yes, rollback and restart later.
  - **Snapshot isolation:**
    - The transaction only sees and operate on a snapshot (copied state) of the DB taken at some time (i.e., before transaction starts), this prevents dirty reads, phantom read, and non-repeatable read.
      - The snapshot copies only committed values for data item.
      - Used by some RDBMSs such as PostgreSQL and Oracle.

Lecture 10 (week11)

- **Distributed system**
    - Interconnected (nodes = hosts = sites) that cooperate in performing a task, by partitioning it into smaller tasks that are solved efficiently in a coordinated manner.
    - Used in the field of Big data (systems that deal with huge and complex datasets)
- **Distributed Database (DDB):**
    - Set of logically connected databases stored across multiple network-connected computers
        - These computers don't have to run the same systems or have the same hardware.
    - Appears to applications and end-users as a single database.
    - Physical location of data (not known to users) affects performance, concurrency, and recovery.
- **Characteristics of DDBs**

1. Transparency (abstraction) *Hiding implementation details (data organization, data replication, data fragmentation) from end-users and applications.*		
Data organization	**Data replication** $(*)$	**Data fragmentation (partitioning)**
Network architecture, physical location of data	Stored copies of data across different hosts for availability, reliability, and read performance. - **Full replication:** a copy of a DB fragment (e.g., table) is present on all hosts. - **Partial replication:** a copy of a DB fragment is present on some of the hosts.	Chopping data across hosts for cloud computing systems and big data. - **Horizonal fragmentation (sharding):** distributing relation **records** (rows) across hosts based on some attribute value or other mechanism. - **Vertical fragmentation:** distributing relation **attributes** (columns) across hosts.

2. Availability and reliability *Most common advantages of DDBs.*	
Availability	**Reliability**
The probability that the system stays available during some time interval (T); percentage of time the system is available. $\textbf{Availability} = \left(\frac{\text{Actual operation time}}{\text{T}}\right) * 100 = \left(1 - \frac{\text{total downtime}}{\text{T}}\right) * 100$	The probability of system to work without disruptions or downtime during some time interval (T) $\textbf{Mean Time Between Failures (MTBF)} = \frac{\text{T} - \text{total downtime}}{\text{number of downtimes}}$
Causes for system failure: - Failure of site (node) - Loss of messages (due to failure of communication link) - Network partitioning (splitting network into disconnected subnets)	**To ensure system reliability:** - Exhaustive system design - Extensive quality control and testing - Fault tolerance: design mechanisms that detect and recover from failures with minimal downtime.

3. Scalability *The limit to which the system can expand (capacity) without interruptions (failures).*	
Horizonal scaling	**Vertical scaling**
Expanding the number of nodes in a distributed system (more nodes = less workload per node)	Expanding (upgrading) a single node (e.g., increasing capacity, improving processing power).

4. Partition tolerance
The system should continue to operate when the network is partitioned (divided into disconnected subsets), however, the communication between them will not be possible.

5. Autonomy (independence)
Each server participating in the distributed database is administered individually. The independence degree of each server (the extent to which the server can operate individually) is called the ***autonomy.***

- (*) **Issues with data replication:**
    - Full replication slows down updates and hinders concurrency control & recovery techniques.
        - One solution is to choose a master primary copy and apply operations only on this copy, that is later propagated (synchronized) whenever necessary.
    - **Design issues:** which fragments to replicate? How many copies to make? How often synchronization is done? Which fragments should be stored in each node? **(data allocation problem)**
        - An optimal/good enough solution to these issues is usually a complex optimization problem.
        - The choice depends on the performance/availability goals and the types/frequencies of transactions submitted at each site.
            - If high availability is required, a **fully redundant DB** might be a good choice.
                - **Fully redundant database:** the entire database is replicated on all hosts.
            - If parts of the DB are mostly accessed from specific site, fragments can only be replicated there.
            - If many updates are performed, number of copies/synchronizations to make should be limited.
- **Concurrency control issues with DDB systems**
    - **Dealing with multiple copies of data:** system should be consistent.
        - **Distinguished (primary) copy:** one copy of the data item is used as the concurrency control coordinator, all lock/unlock requests on that data item are sent to this site only.

Primary site technique	Primary copy technique
• All primary copies are stored at the same primary site. • Introduces a bottleneck of the DDB. • A backup primary site can be used to recover from possible failure of the primary site.	• Primary copies are stored in different sites. • Distributes the lock load. • A new coordinator for the data item must be chosen when the primary one fails.

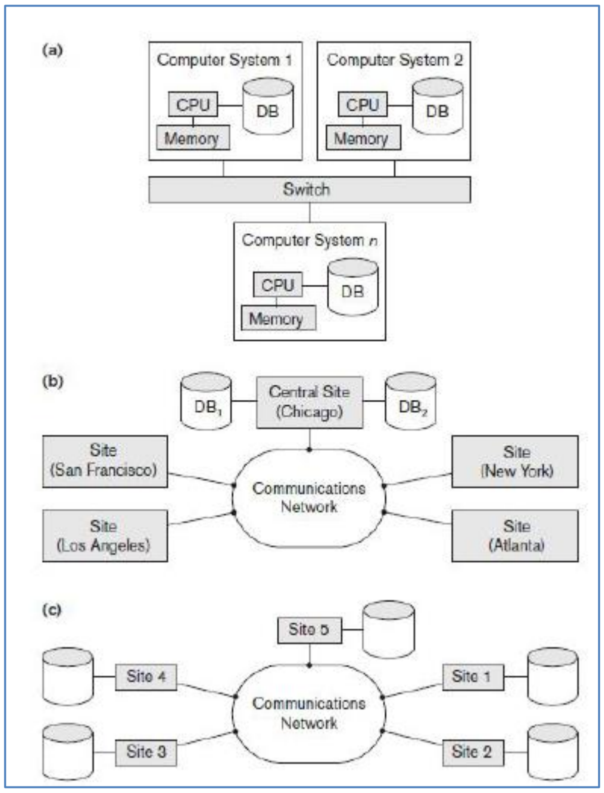
    - **Failure of an individual host:** system should keep operating and recover properly.
        - **Failure of a host during a distributed commit:** two-phase commit is often used to deal with this.
    - **Failure of communication links:**
        - System should keep operating, although not fully if a network partition happened.
    - **Distributed deadlock (among several hosts):** system should resolve the situation.
- **Transaction management in DDB systems**

	Transaction coordinator	Local transaction manager	Global transaction manager
Location	- At each site.	- At each site.	- One site that may change, it can be the one from which the transaction originated.
Respon-sibilities	- Starting transactions originating from this site - Distributing sub-transactions on sites. - Coordinating the termination (all committing or all aborting)	- Maintaining a log for recovery. - Coordinating concurrent execution of transactions at that site.	- Coordinate transaction execution across transaction managers. - Ensures that updates are reflected on all sites. - Ensures that all sites either commit or abort.

- **Taxonomy (classification) of DDBs**

	Traditional centralized DB	Pure DDB	Federated DB	Peer to peer (multi) DB
Autonomy	✓	✗	✓ (to some extent)	✓
Distribution	✗	✓	✓	✓
Heterogeneity	✗	✗	✓	✓

  - **Federated Database System (FDBS):** virtual abstraction of multiple autonomous DBs as a single federated system with a global view, it enables applications to send one query that is handled by multiple different DBMSs running on different hosts.
  - **Heterogeneous DB:** non-homogeneous; sites doesn't have the same DB architecture and/or don't run the same DBMS.
- **DDB architectures**
  - **(a) Parallel database architecture**
    - Multiple resources (CPUs, Memories, Disks) are used together to perform DB operations in parallel.
    - This can be achieved by having multiple nodes **physically close** to each other and connected through a high-speed private LAN.
      - They can share **memory**, **disk**, or **nothing** at all (shared-nothing architecture).
      - The database itself can be distributed or replicated across nodes.
    - Fast, reliable, small communication overhead.
  - **(b) Centralized database with distributed access**
    - The DB is stored in a single location, multiple remote sites can access it.
  - **(c) Purely distributed database**
    - The DB is divided across different remote sites that communicate through a public network.



- **Database classification based on server architecture:**
  - **Client/Server architecture**
    - DB users connect directly to the server.
    - Direct queries only.
  - **Client/Collaborative-servers architecture**
    - DB users connect to one server for queries.
    - That server can then act as a client to another server from the set.
    - Supports direct/indirect queries.

Direct query: executed by one server.
Indirect query: executed by multiple collaborative servers.

  - **Peer-to-peer architecture**
    - All nodes have the same role and functionality, each can act as a server or client.
    - Scalable, flexible, but harder to manage.
- **Distributed catalog management**
  - DDB catalog contains metadata about the database system.
  - Catalogs are critical for performance, they ensure site autonomy, view management, data distribution and replication.
  - **Management schemas for distributed catalogs**
    - **Centralized catalog:** entire catalog stored in one site.
    - **Fully replicated catalog:** entire catalog stored in all sites.
    - **Partitioned catalog:** each site contains catalogs only for its data, updates are tracked by system and propagated immediately.

Lecture 11 (week12)

__SQL databases__ typically store data in a tabular form. (i.e., relation (2D table), or OLAP cube/hypercube (n-dimensional table)).
__NoSQL databases__ use a different storage scheme.

- **NoSQL (Not only SQL)**
  - Databases (typically distributed) that provide a non-tabular, non-relational system for storing and retrieving semi-structured data.
  - **Advantages:** high availability, high scalability, fast performance, big data capability, easy replication.
  - **Disadvantages:** less emphasize on immediate data consistency, query language enhancement, structuring data.

- **NoSQL for DDBMSs**
  - **Replication models**
    - **Master-slave:** master copy is updated; changes propagate to slaves.
    - **Master-master:** any copy can be updated; changes propagate to others.
      - Not guaranteed to have the same copy at all nodes at a time.
      - Data reconciliation methods needed to handle write conflicts.

Advantages	Disadvantages
▪ **Horizontal scalability (file sharding):** generally used to distribute the load and improve availability. ▪ **High performance data access:** using key hashing (to find the record location), or range partitioning (record location is determined by the range to which the key belong to, similar to the idea of b-tree) ▪ **No schema required:** constraints are specified by the application accessing the DB in the format of JSON or XML files ▪ **Versioning:** most NoSQL DBs allow storing multiple versions of the DB for recovery.	▪ **Lower performance (high overhead)** if serializable (strongest level) consistency is required. ▪ **Less powerful query language:** not all SQL queries can be mapped to NoSQL queries. • **Example:** JOIN operation is not available in many NoSQL system and must be implemented in the application) ▪ Any distributed database system will have a disadvantage due to CAP theorem (discussed below)

  - **CAP theorem:**
    - It is impossible for a DDB system to provide data **C**onsistency, **A**vailability, and **P**artition-tolerance simultaneously.
      - **Consistency:** every read request gets the latest data or an error.
      - **Availability:** every request receives a non-error response.
      - **Partition tolerance:** the system continues to operate even if the network is partitioned (divided into disconnected subnets)
- **NoSQL categories**

Document-based storage	Key-value storage
• Data is stored as a **collection** of **documents** (e.g., JSON files) • A document has a unique id, possibly a sophisticated index. • Documents are self-describing (no schema required) • **Examples:** CouchDB, MongoDB	• Data is stored as **values** in a hash-map and accessed by unique keys. • Values can be records (i.e., tuples), objects, documents (e.g., JSON), or a byte-stream to be interpreted by the application. • No query language, rather a set of operations to be used by application programmers. • **Examples:** Redis, Riak, Oracle NoSQL.

Column-based storage	Graph storage
• Data is stored as column families (files, each containing a subset of the original table columns, each called a column qualifier) • Data is queried using a multi-dimensional key. ○ Typically, a tuple (table name, row PK value, column family, column qualifier, timestamp). • **Examples:** ○ Apache HBase (uses Hadoop Distributed File System (HDFS) and Amazon S3 for storage) ○ Google BigTable (uses Google File System (GFS) storage) ○ Apache Cassandra, Yandex ClickHouse.	• Data is stored and represented as a graph (collection of (labeled) nodes/vertices representing entities, and edges representing relations between entities) • Related nodes can be found by traversing edges using path expressions. • **Examples:** Neo4J, ArangoDB, TerminusDB, InfiniteGraph.

- **Case studies**
  - **MongoDB:** document-based storage, provides consistency and partition tolerance.
    - MongoDB stores data in Binary JSON (BSON) files that allow additional data types and is more efficient.
    - A typical deployment scenario for a distributed MongoDB consists of **shards** (units of data) that are connected to **config files** (containing collections metadata) via **routers**. Multiple shard servers form a **replica set**.
    - Consistency and partition tolerance are ensured by making the database unavailable during network partitioning (CAP theorem)
    - MongoDB reads/write preferences:
      - **Default:** read from primary shard server (guarantee strong consistency)
      - **Alternative:** read from secondary server (eventual consistency)
  - **Amazon DynamoDB:** supports both, document-based and key-value storage.
    - For a distributed DynamoDB with many machines, when a new one joins the cluster, a random number is assigned to it that is used to distribute the load between machines.
      - Let the system contain only one machine A, it stores all values for keys ∈ [0, mx]
      - Another machine is added with random number x, now A stores $[0, mx - x]$, B stores $[x, mx]$.
  - **Neo4J:** graph-based storage, neo4j platform provides 3 main features.
    - **Availability** through fault tolerance for transaction processing
      - Platform will remain available as long as the majority of servers are running.
    - **Scalability:** complex read queries are done efficiently through read replicas.
    - **Casual consistency:** a client application is guaranteed to read at least its own writes.
- **NewSQL**
  - A modern-architecture, modern-capabilities DBMSs that provides:
    - The scalability, availability, performance, flexibility provided by NoSQL.
    - The support for SQL queries and ACID transactions that NoSQL usually omits.
  - They are not a replacement for SQL, but rather an improvement.
    - We still need relational model, single row consistency, and ACID transactions.
    - We need immediate data consistency not provided by NoSQL AP (from CAP) DBs.
  - **Examples:** VoltDB, Clusterix, MemSQL, Translattice.

- **Distributed SQL** (subset of NewSQL)
  - Relational distributed databases that take advantage of cloud systems and can work on unreliable infrastructure.
  - **Examples:** Google Spanner, YugabyteDB (free, open source, high availability, handles large amount of distributed data, single-digit latency (<10ms reads))

Lecture 12 (week14)

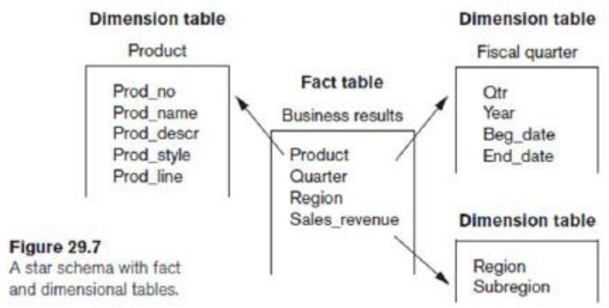
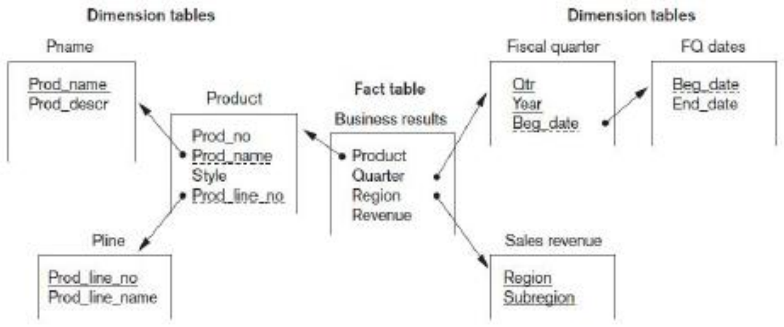
> "***Data mining*** *is a process of extracting and discovering patterns in large data sets. It involves methods at the intersection of* ***machine learning***, ***statistics***, *and* ***database systems***"*.*

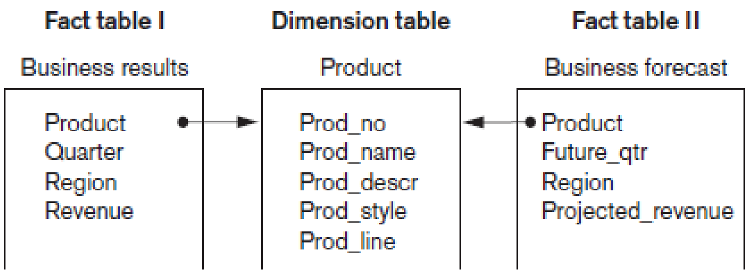
- **OLAP vs OLTP**

On-Line Analytical Processing	On-Line Transaction Processing
A system used for data analysis and decision support; it provides summary reports for decision makers regarding an application/DB.	A system used by information-system experts that manages a transaction-oriented application/DB.
Applies complex (MDX, XQuery, XMLA, …) queries to a system (e.g., a data warehouse) to extract useful information from aggregated (archived, historical) data.	Applies simple (SQL, QBE, QUEL, …) CRUD queries to the system (used with traditional transaction oriented DBs)
Data is updated and normalized on a regular basis (not in real-time)	Data is kept up-to-date and normalized frequently.
Process big amount of data (Tera-Zeta) bytes.	Data is typically not very big (Mega-Giga bytes)

- **Data Warehouse (DW):**
  - A large repository used for storing **historical data** from various operational systems in one place.
  - This data is typically analyzed by means of OLAP systems to provide summarized reports.
  - These reports are then used to support taking informed business decisions, which are typically applied to the databases by means of OLTP systems.
  - DW data characteristics
    - **Integrated:** combined and aggregated from various sources, then stored in a multidimensional fashion.
      - **DW data insertion** is done by:
        - Collecting the data into an **Operational Data Store (ODS).**
        - **Extraction, Transformation, and Loading (ETL)** process of data into the DW.
          - **Loading policies:** append, partial refreshing, sliding window.
      - **Querying:** parametric queries, ad-hoc queries, data mining.
    - **Time-variant:** the data is updated on a regular basis (refresh policy), no real-time analysis.
    - **Non-volatile:** can't be altered (i.e., read/append/purge-all) only.
    - **Subject oriented**: provides information about a topic for decision making rather than an ongoing operation.
  - **Data modeling (storage schemas) for DWs**
    - **Data cube model**
      - **OLAP cube (3D) or hypercube (n-d):** multidimensional array of data optimized for handling multiple data dimensions (attributes) efficiently.
      - **Basic data cube operations [explanation]**
        - Roll-up and Drill-down, Slice and dice, Pivot (rotate)
      - **Example OLAP tools:** IBM Cognos, MicroStrategy, Pentaho, Apache Kylin.
    - **Multidimensional model:** stores the DW as a relational database in some RDBMS.
      - **Dimension:** table of attributes (a normal relation)
      - **Fact:** table of pointers to dimensions (a table of relations)
        - Name comes from the way we gather fact table entries by observing facts.

> **Data mart:** "*data warehouse with a narrower scope*" (i.e., focusing on a small subset of data relating to a specific department for example).
>
> **Data lake:** *water (data) flows from streams (data sources) into the* ***"data lake"****, then collected (structured) into bottles (data marts) for easy consumption.*

Star schema	Snowflake schema
One fact directly referencing all dimensions (typically small tables that are used with bitmap indexes to improve performance).	One fact referencing dimensions that may reference smaller dimensions.
 Dimension table Product Prod_no Prod_name Prod_descr Prod_style Prod_line Fact table Business results Product Quarter Region Sales_revenue Dimension table Fiscal quarter Qtr Year Beg_date End_date Dimension table Region Subregion **Figure 29.7** A star schema with fact and dimensional tables.	 Dimension tables Pname Prod_name Prod_descr Product Prod_no Prod_name Style Prod_line_no Pline Prod_line_no Prod_line_name Fact table Business results Product Quarter Region Revenue Dimension tables Fiscal quarter Qtr Year Beg_date FQ dates Beg_date End_date Sales revenue Region Subregion

Fact constellation (Galaxy schema)	
Set of fact tables that share some dimension table.	 Fact table I Business results Product Quarter Region Revenue Dimension table Product Prod_no Prod_name Prod_descr Prod_style Prod_line Fact table II Business forecast Product Future_qtr Region Projected_revenue

- **Data Warehouse vs Database view:**

Data warehouse	Database view
- Data is stored persistently. - Data is stored in multidimensional arrays. - DW can be indexed for performance. - Huge amount of data (bigger that a whole DB) - Data brought from multiple sources and gone through complex ETL process.	- Data is materialized into a view on demand (SELECT). - Views are in 2D. - Views can't be indexed. - Only a subset of DB. - Data is extracted from DB via a predefined query.

- **Difficulties of implementing DWs:**
  - **Operational issues:** construction, administration, quality control and project management.
    - **Administration**
      - Proportional to the size and complexity of the DW.
      - Requires wide set of technical and non-technical skills (knowledge of business rules, regulations, and constraints, careful coordination, and effective leadership) than needed for a traditional DBA.