# IT344- Database Management Systems

**Study Guide for Midterm Exam**

**College of Computing and Informatics**
**Saudi Electronic University**

# Chapter 17

Disk Storage, Basic File Structure and Hashing

## Disk Storage Devices

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A **disk pack** contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular **tracks** on each disk **surface**.
  - Track capacities vary typically from 4 to 50 Kbytes or more

## Disk Storage Devices (cont.)

- A track is divided into smaller **blocks** or **sectors**
  - because it usually contains a large amount of information
- The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed.
  - One type of sector organization calls a portion of a track that subtends a fixed angle at the center as a sector.
- A track is divided into **blocks**.
  - The block size B is fixed for each system.
    - Typical block sizes range from B=512 bytes to B=4096 bytes.
  - Whole blocks are transferred between disk and main memory for processing.

## Disk Storage Devices (cont.)

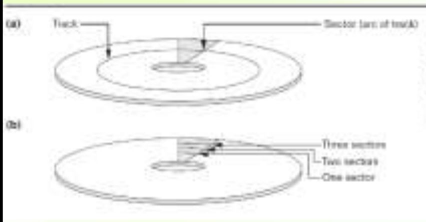


**Figure 17.2**
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

## Disk Storage Devices (cont.)

- A **read-write head** moves to the track that contains the block to be transferred.
  - Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
  - a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)
  - the track number or surface number (within the cylinder)
  - and block number (within track).
- Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) **rd**.
- Double buffering can be used to speed up the transfer of contiguous disk blocks.

## Records

- Fixed and variable length records
- Records contain fields which have values of a particular type
  - E.g., amount, date, time, age
- Fields themselves may be fixed length or variable length
- Variable length fields can be mixed into one record:
  - Separator characters or length fields are needed so that the record can be "parsed."

## Blocking

- **Blocking**:
  - Refers to storing a number of records in one block on the disk.
- Blocking factor (**bfr**) refers to the number of records per block.
- There may be empty space in a block if an integral number of records do not fit in one block.
- **Spanned Records**:
  - Refers to records that exceed the size of one or more blocks and hence span a number of blocks.

## Files of Records

- A **file** is a *sequence* of records, where each record is a collection of data values (or data items).
- A **file descriptor** (or **file header**) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.
- Records are stored on disk blocks.
- The **blocking factor bfr** for a file is the (average) number of file records stored in a disk block.
- Calculation of blocking factor???
- A file can have **fixed-length** records or **variable-length** records.

## Files of Records (cont.)

- File records can be **unspanned** or **spanned**
  - **Unspanned**: no record can span two blocks
  - **Spanned**: a record can be stored in more than one block
- The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.
- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.
- Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**.
  - Usually spanned blocking is used with such files.

## Operation on Files

- Typical file operations include:
  - **OPEN**: Readies the file for access, and associates a pointer that will refer to a *current* file record at each point in time.
  - **FIND**: Searches for the first file record that satisfies a certain condition, and makes it the current file record.
  - **FINDNEXT**: Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
  - **READ**: Reads the current file record into a program variable.
  - **INSERT**: Inserts a new record into the file & makes it the current file record.
  - **DELETE**: Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
  - **MODIFY**: Changes the values of some fields of the current file record.
  - **CLOSE**: Terminates access to the file.
  - **REORGANIZE**: Reorganizes the file records.
    - For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
  - **READ_ORDERED**: Read the file blocks in order of a specific field of the file.

## Unordered Files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
  - This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records.

## Ordered Files

- Also called a **sequential** file.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the correct order.
  - It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its *ordering field* value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

## Average Access Times

- The following table shows the average access time to access a specific record for a given type of file

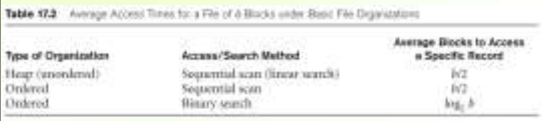


**Table 17.2** Average Access Times for a File of *b* Blocks under Basic File Organizations

| Type of Organization | Access/Search Method | Average Blocks to Access a Specific Record |
|---|---|---|
| Heap (unordered) | Sequential scan (linear search) | $b/2$ |
| Ordered | Sequential scan | $b/2$ |
| Ordered | Binary search | $\log_2 b$ |

## Hashed Files

- Hashing for disk files is called **External Hashing**
- The file blocks are divided into M equal-sized **buckets**, numbered $\text{bucket}_0$, $\text{bucket}_1$, ..., $\text{bucket}_{M-1}$.
  - Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the **hash key** of the file.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the **hashing function**.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together.

## Dynamic And Extendible Hashed Files

- Dynamic and Extendible Hashing Techniques
  - Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records.
  - These techniques include the following: **dynamic hashing, extendible hashing**, and **linear hashing.**
- Both dynamic and extendible hashing use the **binary representation** of the hash value h(K) in order to access a **directory**.
  - In dynamic hashing the directory is a binary tree.
  - In extendible hashing the directory is an array of size $2^d$ where d is called the **global depth**.

## Dynamic And Extendible Hashing (cont.)

- The directories can be stored on disk, and they expand or shrink dynamically.
  - Directory entries point to the disk blocks that contain the stored records.
- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks.
  - The directory is updated appropriately.
- Dynamic and extendible hashing do not require an overflow area.
- Linear hashing does require an overflow area but does not use a directory.
  - Blocks are split in *linear order* as the file expands.

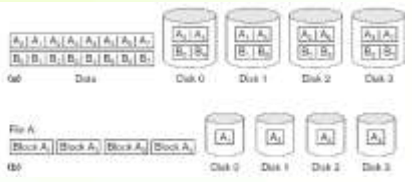
## Parallelizing Disk Access using RAID Technology.

- Secondary storage technology must take steps to keep up in performance and reliability with processor technology.
- A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**.
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.

## RAID Technology (cont.)

- A natural solution is a large array of small independent disks acting as a single higher-performance logical disk.
- A concept called **data striping** is used, which utilizes parallelism to improve disk performance.
- Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk.

Figure 17.13
Striping of data across multiple disks. (a) Bit-level striping across four disks. (b) Block-level striping across four disks.



## RAID Technology (cont.)

- Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.
  - Raid level 0 has no redundant data and hence has the best write performance at the risk of data loss
  - Raid level 1 uses mirrored disks.
  - Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
  - Raid level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed.
  - Raid Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks.
  - Raid level 6 applies the so-called P + Q redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.

## Use of RAID Technology (cont.)

- Different raid organizations are being used under different situations
  - Raid level 1 (mirrored disks) is the easiest for rebuild of a disk from other disks
    - It is used for critical applications like logs
  - Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components.
    - Level 2 includes both error detection and correction.
  - Raid level 3 (single parity disks relying on the disk controller to figure out which disk has failed) and level 5 (block-level data striping) are preferred for Large volume storage, with level 3 giving higher transfer rates.
- Most popular uses of the RAID technology currently are:
  - Level 0 (with striping), Level 1 (with mirroring) and Level 5 with an extra drive for parity.
- Design Decisions for RAID include:
  - Level of RAID, number of disks, choice of parity schemes, and grouping of disks for block-level striping.

## Storage Area Networks

- The demand for higher storage has risen considerably in recent times.
- Organizations have a need to move from a static fixed data center oriented operation to a more flexible and dynamic infrastructure for information processing.
- Thus they are moving to a concept of Storage Area Networks (SANs).
  - In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.
- This allows storage systems to be placed at longer distances from the servers and provide different performance and connectivity options.

# Chapter 18

Indexing Structures for Files

## Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries **<field value, pointer to record>**, which is ordered by field value
- The index is called an access path on the field.

## Indexes as Access Paths (cont.)

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values

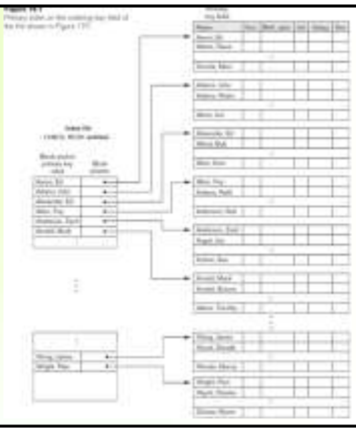
## Indexes as Access Paths (cont.)

- Example: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ... )
- Suppose that:
  - record size R=150 bytes block size B=512 bytes r=30000 records
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks
- For an index on the SSN field, assume the field size $V_{SSN}$=9 bytes, assume the record pointer size $P_R$=7 bytes. Then:
  - index entry size $R_i$=($V_{SSN}$+ $P_R$)=(9+7)=16 bytes
  - index blocking factor $Bfr_i$= B div $R_i$= 512 div 16= 32 entries/block
  - number of index blocks b= (r/ $Bfr_i$)= (30000/32)= 938 blocks
  - binary search needs $\log_2$bI= $\log_2$938= 10 block accesses
  - This is compared to an average linear search cost of:
    - (b/2)= 30000/2= 15000 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2$b= $\log_2$30000= 15 block accesses

## Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a **key field**
  - Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
  - A similar scheme can use the *last record* in a block.
  - A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

## Primary Index on the Ordering Key Field



## Types of Single-Level Indexes

- Clustering Index
  - Defined on an ordered data file
  - The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
  - Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
  - It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index.

## Types of Single-Level Indexes

- Secondary Index
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
  - The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
  - The index is an ordered file with two fields.
    - The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
    - The second field is either a **block** pointer or a record pointer.
    - There can be *many* secondary indexes (and hence, indexing fields) for the same file.
  - Includes one entry *for each record* in the data file; hence, it is a *dense index*

## Properties of Index Types

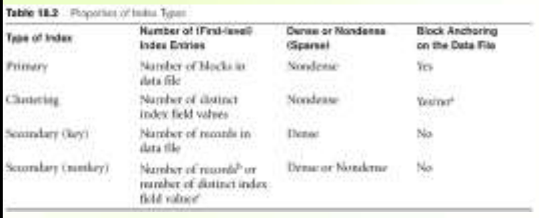


Table 18.2 Properties of Index Types

| Type of Index | Number of (First-level) Index Entries | Dense or Nondense (Sparse) | Block Anchoring on the Data File |
|---|---|---|---|
| Primary | Number of blocks in data file | Nondense | Yes |
| Clustering | Number of distinct index field values | Nondense | Yes/no[a] |
| Secondary (key) | Number of records in data file | Dense | No |
| Secondary (nonkey) | Number of records[b] or number of distinct index field values[c] | Dense or Nondense | No |

## Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

## Multi-Level Indexes

- Such a multi-level index is a form of *search tree*
  - However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.

## Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem
  - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full

## Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

## B-tree Structures

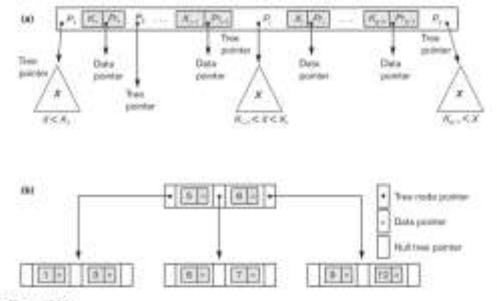


Figure 18.10
B-tree structures. (a) A node in a B-tree with q − 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Chapter 19

Algorithms for Query processing and Optimization

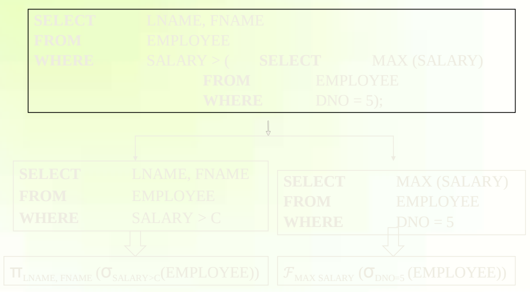
## 0. Introduction to Query Processing (1)

- **Query optimization**:
  - The process of choosing a suitable execution strategy for processing a query.
- Two internal representations of a query:
  - **Query Tree**
  - **Query Graph**

## 1. Translating SQL Queries into Relational Algebra (1)

- **Query block**:
  - The basic unit that can be translated into the algebraic operators and optimized.
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.
- Aggregate operators in SQL must be included in the extended algebra.

## Translating SQL Queries into Relational Algebra (2)



```
SELECT     LNAME, FNAME
FROM       EMPLOYEE
WHERE      SALARY > (   SELECT     MAX (SALARY)
                        FROM       EMPLOYEE
                        WHERE      DNO = 5);
```

```
SELECT     LNAME, FNAME
FROM       EMPLOYEE
WHERE      SALARY > C
```

```
SELECT     MAX (SALARY)
FROM       EMPLOYEE
WHERE      DNO = 5
```

$\pi_{LNAME,\ FNAME}\ (\sigma_{SALARY>C}(EMPLOYEE))$

$\mathcal{F}_{MAX\ SALARY}\ (\sigma_{DNO=5}\ (EMPLOYEE))$

## 7. Using Heuristics in Query Optimization (1)

- Process for heuristics optimization
  1. The parser of a high-level query generates an initial internal representation;
  2. Apply heuristics rules to optimize the internal representation.
  3. A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- The main heuristic is to apply first the operations that reduce the size of intermediate results.
  - E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

## Using Heuristics in Query Optimization (2)

- **Query tree**:
  - A tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as **leaf nodes** of the **tree**, and represents the relational algebra operations as internal nodes.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.
- **Query graph**:
  - A graph data structure that corresponds to a relational calculus expression. It does *not* indicate an order on which operations to perform first. There is only a *single* graph corresponding to each query.

## Using Heuristics in Query Optimization (3)

- Example:
  - For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.
- Relation algebra:

$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}$ $(((\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT}))$ $\bowtie_{\text{DNUM=DNUMBER}}$ $(\text{DEPARTMENT}))$ $\bowtie_{\text{MGRSSN=SSN}}$ $(\text{EMPLOYEE}))$

- SQL query:

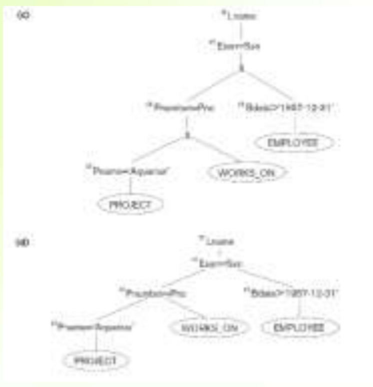
```
Q2:     SELECT     P.NUMBER,P.DNUM,E.LNAME,
                   E.ADDRESS, E.BDATE
        FROM       PROJECT AS P,DEPARTMENT AS D,
                        EMPLOYEE AS E
        WHERE      P.DNUM=D.DNUMBER AND
                   D.MGRSSN=E.SSN AND
                   P.PLOCATION='STAFFORD';
```

## Using Heuristics in Query Optimization (5)

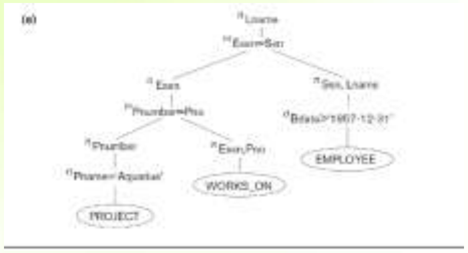
- Heuristic Optimization of Query Trees:
  - The same query could correspond to many different relational algebra expressions — and hence many different query trees.
  - The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.
- Example:

```
Q:  SELECT    LNAME
    FROM      EMPLOYEE, WORKS_ON, PROJECT
    WHERE     PNAME = 'AQUARIUS' AND
              PNMUBER=PNO AND ESSN=SSN
              AND BDATE > '1957-12-31';
```

## Using Heuristics in Query Optimization (7)



## Using Heuristics in Query Optimization (8)



## Using Heuristics in Query Optimization (9)

- General Transformation Rules for Relational Algebra Operations:

1. Cascade of σ: A conjunctive selection condition can be broken up into a cascade (sequence) of individual σ operations:
   - $\sigma_{\text{c1 AND c2 AND ... AND cn}}(R) = \sigma_{c1}(\sigma_{c2}(...(\sigma_{cn}(R))...))$
2. Commutativity of σ: The σ operation is commutative:
   - $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$
3. Cascade of π: In a cascade (sequence) of π operations, all but the last one can be ignored:
   - $\pi_{List1}(\pi_{List2}(...(\pi_{Listn}(R))...)) = \pi_{List1}(R)$
4. Commuting σ with π: If the selection condition c involves only the attributes A1, ..., An in the projection list, the two operations can be commuted:
   - $\pi_{A1, A2, ..., An}(\sigma_c(R)) = \sigma_c(\pi_{A1, A2, ..., An}(R))$

## Using Heuristics in Query Optimization (10)

- General Transformation Rules for Relational Algebra Operations (contd.):

5. Commutativity of ⋈ ( and x ): The ⋈ operation is commutative as is the x operation:
   - $R \bowtie_C S = S \bowtie_C R$; $R \times S = S \times R$
6. Commuting σ with ⋈ (or x ): If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:
   - $\sigma_c(R \bowtie S) = (\sigma_c(R)) \bowtie S$

- Alternatively, if the selection condition c can be written as (c1 and c2), where condition c1 involves only the attributes of R and condition c2 involves only the attributes of S, the operations commute as follows:
  - $\sigma_c(R \bowtie S) = (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$

## Using Heuristics in Query Optimization (11)

- General Transformation Rules for Relational Algebra Operations (contd.):

7. Commuting π with $\bowtie$ (or x): Suppose that the projection list is L = {A1, ..., An, B1, ..., Bm}, where A1, ..., An are attributes of R and B1, ..., Bm are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:
   - $\pi_L ( R \bowtie_C S ) = (\pi_{A1, ..., An} (R)) \bowtie_C (\pi_{B1, ..., Bm} (S))$
- If the join condition C contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed.

## Using Heuristics in Query Optimization (12)

- General Transformation Rules for Relational Algebra Operations (contd.):

8. Commutativity of set operations: The set operations ∪ and ∩ are commutative but "–" is not.
9. Associativity of $\bowtie$, x, ∪, and ∩ : These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have
   - $( R\ \theta\ S )\ \theta\ T = R\ \theta\ ( S\ \theta\ T )$
10. Commuting σ with set operations: The σ operation commutes with ∪ , ∩ , and –. If θ stands for any one of these three operations, we have
   - $\sigma_c ( R\ \theta\ S ) = (\sigma_c (R))\ \theta\ (\sigma_c (S))$

## Using Heuristics in Query Optimization (15)

- Summary of Heuristics for Algebraic Optimization:
  1. The main heuristic is to apply first the operations that reduce the size of intermediate results.
  2. Perform select operations as early as possible to reduce the number of tuples and perform project operations as early as possible to reduce the number of attributes. (This is done by moving select and project operations as far down the tree as possible.)
  3. The select and join operations that are most restrictive should be executed before other similar operations. (This is done by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree appropriately.)

## Using Heuristics in Query Optimization (16)

- Query Execution Plans
  - An execution plan for a relational algebra query consists of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.
  - **Materialized evaluation**: the result of an operation is stored as a temporary relation.
  - **Pipelined evaluation**: as the result of an operator is produced, it is forwarded to the next operator in sequence.

## 8. Using Selectivity and Cost Estimates in Query Optimization (1)

- **Cost-based query optimization**:
  - Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.
  - (Compare to heuristic query optimization)
- Issues
  - Cost function
  - Number of execution strategies to be considered

## Using Selectivity and Cost Estimates in Query Optimization (2)

- Cost Components for Query Execution
  1. Access cost to secondary storage
  2. Storage cost
  3. Computation cost
  4. Memory usage cost
  5. Communication cost
- Note: Different database systems may focus on different cost components.

## Using Selectivity and Cost Estimates in Query Optimization (3)

- Catalog Information Used in Cost Functions
  - Information about the size of a file
    - number of records (tuples) (r),
    - record size (R),
    - number of blocks (b)
    - blocking factor (bfr)
  - Information about indexes and indexing attributes of a file
    - Number of levels (x) of each multilevel index
    - Number of first-level index blocks (bI1)
    - Number of distinct values (d) of an attribute
    - Selectivity (sl) of an attribute
    - Selection cardinality (s) of an attribute. (s = sl * r)

## Using Selectivity and Cost Estimates in Query Optimization (4)

- Examples of Cost Functions for SELECT
- S1. Linear search (brute force) approach
  - $C_{S1a} = b$;
  - For an equality condition on a key, $C_{S1a} = (b/2)$ if the record is found; otherwise $C_{S1a} = b$.
- S2. Binary search:
  - $C_{S2} = \log_2 b + (s/bfr)\rceil - 1$
  - For an equality condition on a unique (key) attribute, $C_{S2} = \log_2 b$
- S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record
  - $C_{S3a} = x + 1$; $C_{S3b} = 1$ for static or linear hashing;
  - $C_{S3b} = 1$ for extendible hashing;

## Using Selectivity and Cost Estimates in Query Optimization (10)

- **Multiple Relation Queries and Join Ordering**
  - A query joining n relations will have n-1 join operations, and hence can have a large number of different join orders when we apply the algebraic transformation rules.
  - Current query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees.

- **Left-deep tree**:
  - A binary tree where the right child of each non-leaf node is always a base relation.
    - Amenable to pipelining
    - Could utilize any access paths on the base relation (the right child) when executing the join.

## 9. Overview of Query Optimization in Oracle

- Oracle DBMS V8
  - **Rule-based query optimization**: the optimizer chooses execution plans based on heuristically ranked operations.
    - (Currently it is being phased out)
  - **Cost-based query optimization**: the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with lowest estimate cost.
    - The query cost is calculated based on the estimated usage of resources such as I/O, CPU and memory needed.
  - Application developers could specify hints to the ORACLE query optimizer.
  - The idea is that an application developer might know more information about the data.

# Chapter 20

# Physical Database Design and Tuning

## 1. Physical Database Design in Relational Databases (1)

- Factors that Influence Physical Database Design:
  A. **Analyzing the database queries and transactions**
  - For each query, the following information is needed.
    1. The *files* that will be accessed by the query;
    2. The *attributes* on which any *selection* conditions for the query are specified;
    3. The *attributes* on which any *join* conditions or conditions to link multiple tables or objects for the query are specified;
    4. The *attributes* whose values will be *retrieved* by the query.
  - Note: the attributes listed in items 2 and 3 above are candidates for definition of access structures.

## Physical Database Design in Relational Databases (2)

- Factors that Influence Physical Database Design (cont.):
  A. **Analyzing the database queries and transactions** (cont.)
  - For each **update** transaction or operation, the following information is needed.
    1. The files that will be updated;
    2. The type of operation on each file (insert, update or delete);
    3. The attributes on which selection conditions for a delete or update operation are specified;
    4. The attributes whose values will be changed by an update operation.
  - Note: the attributes listed in items 3 above are candidates for definition of access structures. However, the attributes listed in item 4 are candidates for avoiding an access structure.

## Physical Database Design in Relational Databases (3)

- Factors that Influence Physical Database Design (cont.):
  B. **Analyzing the expected frequency of invocation of queries and transactions**
  - The expected frequency information, along with the attribute information collected on each query and transaction, is used to compile a cumulative list of expected frequency of use for all the queries and transactions.
  - It is expressed as the expected frequency of using each attribute in each file as a selection attribute or join attribute, over all the queries and transactions.
  - **80-20 rule**
    - 20% of the data is accessed 80% of the time

## Physical Database Design in Relational Databases (4)

- Factors that Influence Physical Database Design (cont.)
  C. **Analyzing the time constraints of queries and transactions**
  - Performance constraints place further priorities on the attributes that are candidates for access paths.
  - The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structure.

## Physical Database Design in Relational Databases (4)

- Factors that Influence Physical Database Design (cont.)
  D. **Analyzing the expected frequencies of update operations**
  - A minimum number of access paths should be specified for a file that is updated frequently.

## Physical Database Design in Relational Databases (4)

- Factors that Influence Physical Database Design (cont.)
  E. **Analyzing the uniqueness constraints on attributes**
  - Access paths should be specified on all candidate key attributes — or set of attributes — that are either the primary key or constrained to be unique.

## Physical Database Design in Relational Databases (5)

- Physical Database Design Decisions
- Design decisions about indexing
  - Whether to index an attribute?
  - What attribute or attributes to index on?
  - Whether to set up a clustered index?
  - Whether to use a hash index over a tree index?
  - Whether to use dynamic hashing for the file?

## Physical Database Design in Relational Databases (6)

- Physical Database Design Decisions (cont.)
- Denormalization as a design decision for speeding up queries
  - The goal of normalization is to separate the logically related attributes into tables to minimize redundancy and thereby avoid the update anomalies that cause an extra processing overheard to maintain consistency of the database.
  - The goal of denormalization is to improve the performance of frequently occurring queries and transactions. (Typically the designer adds to a table attributes that are needed for answering queries or producing reports so that a join with another table is avoided.)
  - Trade off between update and query performance

## 2. An Overview of Database Tuning in Relational Systems (1)

- **Tuning**:
  - The process of continuing to revise/adjust the physical database design by monitoring resource utilization as well as internal DBMS processing to reveal bottlenecks such as contention for the same data or devices.
- Goal:
  - To make application run faster
  - To lower the response time of queries/transactions
  - To improve the overall throughput of transactions

## An Overview of Database Tuning in Relational Systems (3)

- Problems to be considered in tuning:
  - How to avoid excessive lock contention?
  - How to minimize overhead of logging and unnecessary dumping of data?
  - How to optimize buffer size and scheduling of processes?
  - How to allocate resources such as disks, RAM and processes for most efficient utilization?

## An Overview of Database Tuning in Relational Systems (7)

- Tuning Queries
  - Indications for tuning queries
    - A query issues too many disk accesses
    - The query plan shows that relevant indexes are not being used.

## An Overview of Database Tuning in Relational Systems (8)

- Tuning Queries (cont.): Typical instances for query tuning
  - In some situations involving using of correlated queries, temporaries are useful.
  - If multiple options for join condition are possible, choose one that uses a clustering index and avoid those that contain string comparisons.
  - The order of tables in the FROM clause may affect the join processing.
  - Some query optimizers perform worse on nested queries compared to their equivalent un-nested counterparts.
  - Many applications are based on views that define the data of interest to those applications. Sometimes these views become an overkill.

# Chapter 21

# Introduction to Transaction Processing Concepts and Theory

## 1 Introduction to Transaction Processing (1)

- **Single-User System**:
  - At most one user at a time can use the system.
- **Multiuser System**:
  - Many users can access the system concurrently.
- **Concurrency**
  - **Interleaved processing**:
    - Concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**:
    - Processes are concurrently executed in multiple CPUs.

## Introduction to Transaction Processing (2)

- A **Transaction**:
  - Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
- A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries**:
  - Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

## Introduction to Transaction Processing (3)

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database** is a collection of named data items
- **Granularity** of data - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic operations are **read** and **write**
  - **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
  - **write_item(X)**: Writes the value of program variable X into the database item named X.

## Introduction to Transaction Processing (6)

Why Concurrency Control is needed:

- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 21.1.4).
  - The updated item is accessed by another transaction before it is changed back to its original value.
- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

## Introduction to Transaction Processing (12)

Why **recovery** is needed:
(What causes a Transaction to fail)

1. A computer failure (system crash):
2. 2. A transaction or system error:
3. 3. Local errors or exception conditions detected by the transaction:
4. 4. Concurrency control enforcement:

5. Disk failure:
6. Physical problems and catastrophes:

## 2 Transaction and System Concepts (1)

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

## Transaction and System Concepts (2)

- Recovery manager keeps track of the following operations:
  - **begin_transaction**: This marks the beginning of transaction execution.
  - **read** or **write**: These specify read or write operations on the database items that are executed as part of a transaction.
  - **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
    - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

## Transaction and System Concepts (3)

- Recovery manager keeps track of the following operations (cont):
  - **commit_transaction**: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
  - **rollback** (or **abort**): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

## Transaction and System Concepts (4)

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

## Transaction and System Concepts (6)

- The System Log
  - **Log** or **Journal**: The log keeps track of all transaction operations that affect the values of database items.
    - This information may be needed to permit recovery from transaction failures.
    - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
    - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

## Transaction and System Concepts (10)

Commit Point of a Transaction:

- **Definition a Commit Point:**
  - A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
  - Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
  - The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:**
  - Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

## 3 Desirable Properties of Transactions (1)

ACID properties:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions  unnecessary (see Chapter 21).
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

## 4 Characterizing Schedules Based on Recoverability (1)

- **Transaction schedule or history**:
  - When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).
- A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:
  - It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

## Characterizing Schedules Based on Recoverability (2)

Schedules classified on recoverability:

- **Recoverable schedule**:
  - One where no transaction needs to be rolled back.
  - A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.
- **Cascadeless schedule**:
  - One where every transaction reads only the items that are written by committed transactions.

## Characterizing Schedules Based on Recoverability (3)

Schedules classified on recoverability (cont.):

- **Schedules requiring cascaded rollback**:
  - A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- **Strict Schedules**:
  - A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

## 5 Characterizing Schedules Based on Serializability (1)

- Serial schedule:
  - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
    - Otherwise, the schedule is called nonserial schedule.
- Serializable schedule:
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

## Characterizing Schedules Based on Serializability (3)

- Being serializable is <u>not</u> the same as being serial
- Being serializable implies that the schedule is a <u>correct</u> schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

## Characterizing Schedules Based on Serializability (6)

- View equivalence:
  - A less restrictive definition of equivalence of schedules
- View serializability:
  - Definition of serializability based on view equivalence.
  - A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

## Characterizing Schedules Based on Serializability (7)

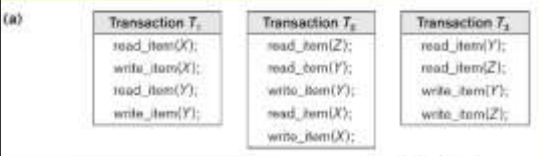
- Two schedules are said to be view equivalent if the following three conditions hold:
  1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
  2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
  3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

## Characterizing Schedules Based on Serializability (11)

**Testing for conflict serializability: Algorithm 21.1:**

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

## Another Example of Serializability Testing

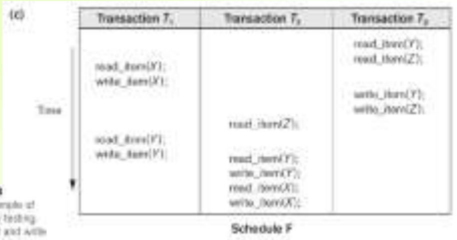


(a)

| Transaction $T_1$ | Transaction $T_2$ | Transaction $T_3$ |
|---|---|---|
| read_item(X); | read_item(Z); | read_item(Y); |
| write_item(X); | read_item(Y); | read_item(Z); |
| read_item(Y); | write_item(Y); | write_item(Y); |
| write_item(Y); | read_item(X); | write_item(Z); |
| | write_item(X); | |

**Figure 21.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule F.

## Another Example of Serializability Testing



(c)

| Time ↓ | Transaction $T_1$ | Transaction $T_2$ | Transaction $T_3$ |
|---|---|---|---|
| | | | read_item(Y); |
| | | | read_item(Z); |
| | read_item(X); | | |
| | write_item(X); | | |
| | | | write_item(Y); |
| | | | write_item(Z); |
| | | read_item(Z); | |
| | read_item(Y); | | |
| | write_item(Y); | | |
| | | read_item(Y); | |
| | | write_item(Y); | |
| | | read_item(X); | |
| | | write_item(X); | |

Schedule F

**Figure 21.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule F.

## Transaction Support in SQL2 (2)

Characteristics specified by a SET TRANSACTION statement in SQL2:

- **Access mode**:
  - READ ONLY or READ WRITE.
    - The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area.

## Transaction Support in SQL2 (3)

Characteristics specified by a SET TRANSACTION statement in SQL2 (cont.):

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
  - With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
  - However, if any transaction executes at a lower level, then serializability may be violated.

## Transaction Support in SQL2 (4)

Potential problem with lower isolation levels:

- **Dirty Read**:
  - Reading a value that was written by a transaction which failed.
- **Nonrepeatable Read**:
  - Allowing another transaction to write a new value between multiple reads of one transaction.
  - A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
    - Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

## Transaction Support in SQL2 (5)

- Potential problem with lower isolation levels (cont.):
  - Phantoms:
    - New rows being read using the same read with a condition.
      - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
      - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
      - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.

## Transaction Support in SQL2 (7)

- Possible violation of serializabilty:

| | Type of Violation | | |
|---|---|---|---|
| Isolation level | Dirty read | nonrepeatable read | phantom |
| READ UNCOMMITTED | yes | yes | yes |
| READ COMMITTED | no | yes | yes |
| REPEATABLE READ | no | no | yes |
| SERIALIZABLE | no | no | no |

# Chapter 22

# Concurrency Control Techniques

## Database Concurrency Control

- 1 Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.
- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

## Database Concurrency Control

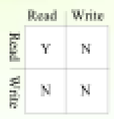
Two-Phase Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

## Database Concurrency Control

Two-Phase Locking Techniques: Essential components

- Two locks modes:
  - (a) shared (read) (b) exclusive (write).
- Shared mode: shared lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix

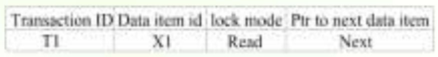


| | Read | Write |
|---|---|---|
| Read | Y | N |
| Write | N | N |

## Database Concurrency Control

Two-Phase Locking Techniques: Essential components

- Lock Manager:
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.



| Transaction ID | Data item id | lock mode | Ptr to next data item |
|---|---|---|---|
| T1 | X1 | Read | Next |

## Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two Phases:
  - (a) Locking (Growing)
  - (b) Unlocking (Shrinking).
- **Locking (Growing) Phase:**
  - A transaction applies locks (read or write) on desired data items one at a time.
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time.
- **Requirement:**
  - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

## Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
  - (a) **Basic**
  - (b) **Conservative**
- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict**:
  - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

## Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock**

| T'1 | T'2 | |
|---|---|---|
| read_lock (Y);<br>read_item (Y); | | T1 and T2 did follow two-phase policy but they are deadlock |
| | read_lock (X);<br>read_item (Y); | |
| write_lock (X);<br>(waits for X) | write_lock (Y);<br>(waits for Y) | |

- Deadlock (T'1 and T'2)

## Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock prevention**
  - A transaction locks all data items it refers to before it begins execution.
  - This way of locking prevents deadlock since a transaction never waits for a data item.
  - The conservative two-phase locking uses this approach.

## Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock detection and resolution**
  - In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
  - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle. One of the transaction o

## Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock avoidance**
  - There are many variations of two-phase locking algorithm.
  - Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

## Database Concurrency Control

Timestamp based concurrency control algorithm

- **Timestamp**
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
  - Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

## Database Concurrency Control

Multiversion concurrency control techniques

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effect:
  - Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

## Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

- Concept
  - Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T.
  - This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X.

## Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

- Steps
  1. X is the committed version of a data item.
  2. T creates a second version X' after obtaining a write lock on X.
  3. Other transactions continue to read X.
  4. T is ready to commit so it obtains a certify lock on X'.
  5. The committed version X becomes X'.
  6. T releases its certify lock on X', which is X now.

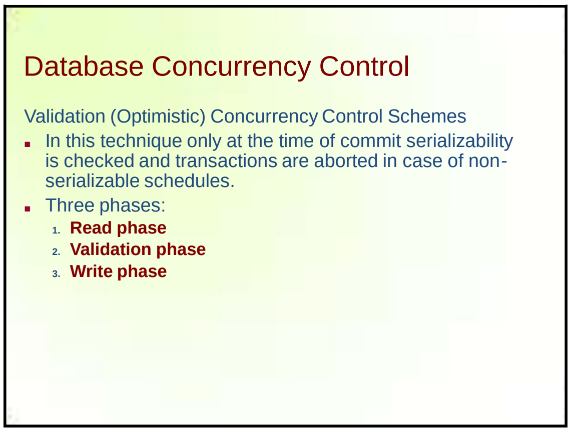
Compatibility tables for

| | Read | Write |
|---|---|---|
| Read | yes | no |
| Write | no | no |

read/write locking scheme

| | Read | Write | Certify |
|---|---|---|---|
| Read | yes | no | no |
| Write | no | no | no |
| Certify | no | no | no |

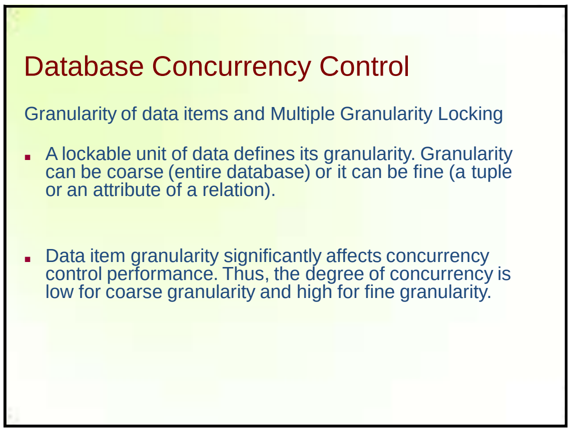
read/write/certify locking scheme



## Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

- In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.
- Three phases:
  1. **Read phase**
  2. **Validation phase**
  3. **Write phase**



## Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation).
- Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity.