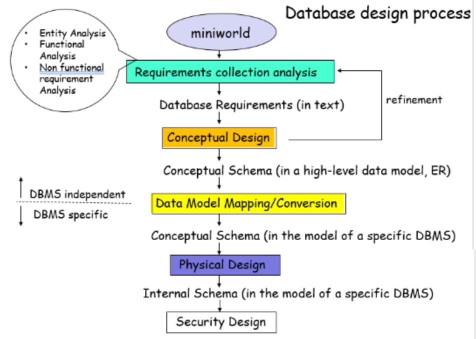
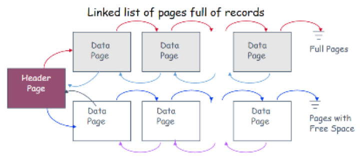
Database design process


Entity Analysis
Functional Analysis
Non functional requirement Analysis
miniworld
Requirements collection analysis
Database Requirements (in text)
refinement
Conceptual Design
Conceptual Schema (in a high-level data model, ER)
DBMS independent
DBMS specific
Data Model Mapping/Conversion
Conceptual Schema (in the model of a specific DBMS)
Physical Design
Internal Schema (in the model of a specific DBMS)
Security Design

One implementation of a Heap File as a List



- The header page id and heap file name must be stored in some place.
- Each page contains at least 2 `pointers' plus data.
- Pointers here are disk block locations.

In most cases, we want DBMS to ensure the ACID properties
Most relational DBMSes do, but some No-SQL DBMS don't

ACID Properties of Transactions

- ATOMICITY: All actions in a transaction are carried out or none are.
- CONSISTENCY: Each transaction with no concurrent execution of other transactions must preserve the consistency of the database. (Developers have to ensure correctness of the program).
- ISOLATION: Transactions are isolated from the effects of other concurrently executing transactions.
- DURABILITY: Once the transaction has been successfully completed, its effects should persist if the system crashes before all its changes are reflected on disk.

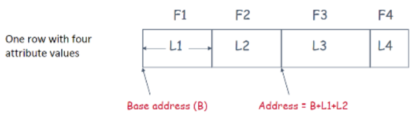
JDBC-Java Database Connection- things needed: setAutoCommit(bool) (false for HW3), setTransactionIsolation, commit(), prepareCall(String), <- Callable Statement <- tied to queries <-stored procedures, Connection

To avoid dirty read and unrepeatable read, use

Connection.Transaction_Repeatable_Read

Fixed Length Row/Record Formats

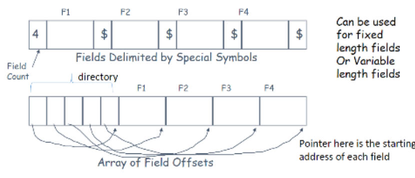
- Each record in the same relation takes the same amount of space.
  Ex: emp(eid integer, name char(40))
  The number of bytes per record is = 4 (size of integer)+40=44.
- Information about field/attribute types is stored in the *system catalog*.
- *Fi: Field/Attribute i;* Li: Size of field i in bytes



Variable Length Record Format

Ex: emp(eid:integer, name:varchar(40)) VARCHAR: Variable length characters

- Two alternative formats (#fields is fixed):


F1
F2
F3
F4
4
$
Fields Delimited by Special Symbols
Field Count
directory
Array of Field Offsets
Can be used for fixed length fields Or Variable length fields
Pointer here is the starting address of each field

Second format offers direct access to the i'th field, efficient storage of *nulls*; but has small directory overhead.

What are file formats used to store a relation?

- Heap File (Unordered file): Suitable when typical access is retrieving all records. Data are not ordered
- Sorted File: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Hashed File: Good for equality selections.

The implementations depend on the underlying DBMS.

SUP_INFO(SNAME,SADDR,ITEM,PRICE)
A, Ames, Basil, 3.99
B, Nevada, Basil, 4.00
B, Nevada, Banana, 0.59
A, Ames, Ground Pork, 2.99

- Problems due to redundancy
  - *Update anomaly*: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
  - *Insertion anomaly*: It may not be possible to store some information unless some other information is stored as well.
  - *Deletion anomaly*: It may not be possible to delete some information without losing some other information as well.

Good relational database design:

1. Reduce redundancy
2. Query performance

T1	T2
	R(X)
	R(Y)
	W(X)
R(X)	
R(Y)	
	W(Y)
W(X)	

RW conflict

3. Give an example schedule write-write conflict.

T1	T2
R(X)	
	R(X)
R(Y)	
	R(Y)
W(X)	
	W(X)
	W(Y)

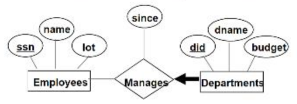
Inferences caused by conflicting actions of different transactions on the same object

- Write Read (WR) Conflicts (dirty reads)
- Read Write (RW) Conflicts (unrepeatable reads)
- Write Write (WW) Conflicts (blind write)

For conflicts to happen, the two actions must be from different transactions and one of them must be a write on the same object!

- Read actions only by different transactions do not create conflicts
- Write actions by different transactions on different objects do not conflict

Entity set with both total constraint and key constraints from one side

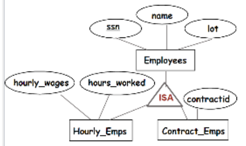


- Handle the key constraint like what we did in the previous slide.
- For total participation, which means that all entities must participate in the relationship set, add NOT NULL to the attribute(s) of the foreign key

Departments(did integer, dname varchar(50), budget float, since date, mgrssn char(11) NOT NULL,

PRIMARY KEY(did),
FOREIGN KEY (mgrssn) references Employees (SSN))

ISA relationship set



- No overlap and no covering constraints
- *Must delete Hourly_Emps tuple if a referenced Employees tuple is deleted.*
- *Must delete Contract_Emps tuple if a referenced Employees tuple is deleted*

3 relation design

- Employees(ssn, name, lot, PRIMARY KEY(ssn))
- *Hourly_Emps* (hssn, *hourly_wages, hours_worked,*
  PRIMARY KEY (hssn),
  FOREIGN KEY (hssn) references Employees(ssn) on delete cascade)
- Contract_Emps(cssn, contractid,
  PRIMARY KEY (cssn),
  FOREIGN KEY(cssn) references Employees (ssn) on delete cascade

Attribute types are omit due to limited space

- The no-covering constraint requires Employees relation since we have to keep information about employees who are not hourly nor contract employees.
- No overlap constraint has to be enforced using a trigger.

There are many types of indexes

- Hash-based indexes based on hashing
- Inverted indexes (value, and pointer(s) to the rows that have the values) like an index at the back of a book.
- Tree-based indexes (ISAM index, B-Tree, B+Tree)
- Spatial indexes for points, regions (R-Tree and variants)

DBMS may choose to implement only some types of indexes
For example, MySQL implements some types of B Tree indexes.

Design Methods to Reduce Redundancy

Two approaches:

1. Design from an Entity-Relationship (ER) diagram and convert them into relational schemas. Refine each schema using functional dependencies (more practical).
2. Design using the theory of relational database design
   - Either put all attributes in one table. Then use the functional dependencies to decompose the table into tables with fewer attributes and with desirable properties (normalization)
     - Must have lossless join and dependency preserving properties
   - *Or start grouping attributes together based on the functional dependencies*

1. Give an example schedule with actions of transac write-read conflict (i.e., a dirty read).

The black horizontal line indicates the end of one

TIME

T1	T2
R(X)	
R(Y)	
W(X)	
	R(X)
	R(Y)
	W(X)
	W(Y)

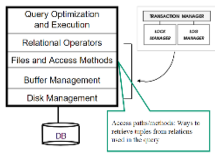
T1	T2	description	step
	S(X)	get shared lock on X	1
	R(X)		
	S(Y)	get shared lock on Y	2
	R(Y)		
	X(X)	get exclusive lock on X	3
	W(X)		
S(X)		request shared lock on X, T2 has exclusive: cannot get	4
~~R(X)~~		deferred	
~~R(Y)~~		deferred	
	X(Y)	get exclusive lock on Y	5
	W(Y)	(releases locks)	6
~~W(X)~~		deferred	
S(X)		get shared lock on X	7
R(X)			
S(Y)		get shared lock on Y	8
R(Y)			
X(X)		get exclusive lock on X	9
W(X)		(releases locks)	

Lock table (subscript indicates: step added, step removed):

object	transactions	Type of lock	Waiting
X_1	~~T2~~$_{1,6}$ T1$_7$	~~shared~~$_{1,3}$ ~~exclusive~~$_{3,6}$ shared$_7$	~~T1~~$_{4,7}$
Y_2	~~T2~~$_{2,6}$ T1$_8$	~~shared~~$_{2,5}$ ~~exclusive~~$_{5,6}$ ~~shared~~$_{8,9}$ exclusive$_9$	

DBMS has a transaction manager to provide protection against interferences and failures

- A typical DBMS has a layered architecture.
- The picture shows one of several possible architectures.
- Each DBMS has its own variations.
- We'll learn the principles and use MySQL as our use case DBMS.
- Use these principles when you work with other relational DBMS


These layers must consider concurrency control and recovery
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Management
DB
TRANSACTION MANAGER
LOCK MANAGER
LOG MANAGER
Access paths/methods: Ways to retrieve tuples from relations used in the query

Strict Two-phase Locking (Strict 2PL) Protocol:

1. Each transaction must obtain a S (*shared*) lock on the object before reading, and an X (*exclusive*) lock on the object before writing.

   If a transaction holds an X lock on an object, it can read the object without additional shared lock required; however, no other transaction can get a lock (S or X) on that object.

2. All locks held by a transaction are released when the transaction completes.

$S_T(O)$: Shared lock on object O
$X_T(O)$: Exclusive lock on object O

What is an index?

- An *index, a data structure that helps* speeding up queries on the *search key attributes*
- Search key: A subset of attributes of a relation that we want to query; we use a frequently used search key to build an index
  - Example: where eid=100; the search key is eid

When DBMS does not consider using indexes

1. When the indexed attribute is used inside a function that is not an aggregate function (min, max, …).

   Ex: where month(posted_date) = 5
   Even there is an index with posted_date as the first attribute, the index won't be considered since posted_date is inside a function month that extracts the value of month from the posted_date attribute with the data type datetime

2. When the indexed attribute is used in a where clause with wildcard matching

   Ex: where name like '%John'

When DBMS does not consider using indexes (Cont'd)

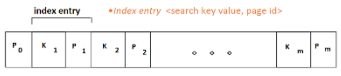
3. When a query involves multiple attributes, but none of these attributes is the first attribute of any of the indexes associated with the relation.

   Why so? We need to understand how an index structure helps speeding up a search

4. When it is more expensive to use relevant indexes than other plans

Search algorithm using index with unique search key values

K: search key value
Output: pointer to a leaf page



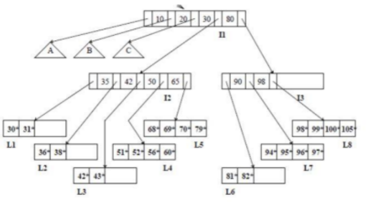
```
find(K) {
   return tree_search(root, K)
}

tree_search(nodepointer, K) {
if (*nodepointer is a leaf), return nodepointer
else
  if (K<K_1) then return tree_search(P_0, K)
  else if (K>=K_m) return tree_search(P_m, K)
  else {
    find i such that K_i<=K<K_i+1
    return tree_search(P_i,K)
  }
}
```

P_i points to a sub-tree in which all key values K in that sub-tree are at least K_i and less than K_{i+1}

Once a leaf page is found, search within the leaf page for the data entry with the search key value of K

Retrieve the page where the desired record is into the database memory buffer pool and search the record in memory.



a. What is the order of this tree? **Order of the trees is the maximum search key value you can store in each node / 2. 4/2 = 2** b. Name all the three nodes that must be fetched to answer the query "Get all the records with a search key value of 38. **I1, I2, L2** c. Name all the three nodes that must be fetched to answer the query "Get all the records with a search key value of 106. **I1, I3, L8** d. Which of these nodes can have less than two key values? **Only the root node can be less than half full. So, I1 e.** What can we say about the content of the subtrees A, B, and C? **B+ trees are self balancing, so the height of the left and right subtree cannot differ by more than one.** dense index has one data entry per record. sparse index has one data entry per page. Each non-root node has d <= m <= 2d entries, where d is the order of the tree. The root node contains 1 <= m <= 2d entries

Relational Operators

- Selection (σ) (sigma) Selects a subset of rows from a relation. This operator needs one input (operand)/
- Projection (π) (pi) Selects a subset of columns from a relation, eliminating duplicates. This operator needs one input.

-*Theta-join* $\bowtie_c$ A join on the condition ***c***

c is a condition (equality, inequality, less than, greater than) or a boolean expression

*Equi-join is the join where **c** contains only **equalities***

- Natural Join ($\bowtie$) Equijoin on *all* fields with the same name; only one copy of the field is kept

Relational operators can be nested!

Examples: Write a relational algebra expression for each of the below queries.

Schemas with data types omitted; attributes of the primary key are underlined.

Emp(eid, ename, salary)
Works(eid, did, pct_time);
eid is the foreign key to Emp (eid) and did is the foreign key to Dept (did).
Dept(did, dname, budget, managerid); Manager id is the foreign key to Emp(eid).

Find eid values of all employees whose name is 'John'.

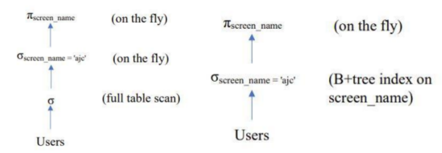
$\pi_{eid}(\sigma_{ename = 'John'}(Emp))$

Find eid, ename values of all employees whose name is 'John' or eid of 101.

$\pi_{eid, ename}(\sigma_{ename = 'John' \text{ or } eid=101}(Emp))$

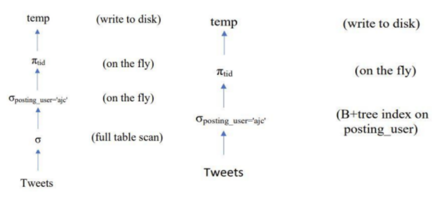
SELECT u.screen_name FROM Users u where u.screen_name = 'ajc'; //relational operators means pi and sigmas

Execution Plan Single Index Access Path Plan



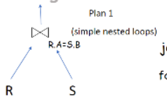
• bytes per record (x) = 1 (character) * 80 + 1 * 80 + 1 * 80 + 1 * 80+ 1 * 80 + 4 (int) + 4 (int) = 408 bytes
• records per page (r) = floor(4000/x) = floor(4000/408) = 9 records due to the assumption that only 4000 bytes are available to store records/tuples
• #total no. of pages for storing all the rows for this relation = ceil(5000/9) = 556 due to assumption that we have 5000 records (or tuples)

INSERT into temp SELECT tid FROM Tweets where posting_user='ajc';



Disk I/O cost = cost of disk I/Os for doing the full table scan + cost of writing the content in temp to disk
• bytes per record = 8 + 4 + 4 + 4 + 4 + 80 = 104 • records per page = floor(4000/104) = 38 rows • total no. of pages for storing all the rows for this relation = ceil(10000/37) = 264 • rows to write to disk = 0.1*10,000 = 1000 rows (using selectivity factor) • bytes per record for temp = 1 * 8 (BigInt) = 8 Bytes • records per page = floor(4000/8) = 500 rows • data pages to write 1000 rows = ceil(1000/500) = 2
• The heap file format needs 1 header page and #pages to keep all the records. However, the assumption asks to ignore the header page, so the disk I/O cost is 264 + 2 = 266

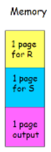
Page at a time Simple Nested Loops Join



join condition R.A=S.B (e.g., Recipe.fid=Food.fid)

```
foreach page in R do // R: outer relation
  foreach page in S do // S: inner relation

    foreach r tuple in the buffer for R
      foreach s tuple in the buffer for S
        if r.A == S.B then add <r, s> to output buffer
        Whenever the output buffer is full, output the result
```



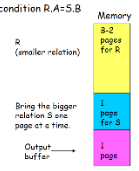
R is the outer relation
S is the inner relation

Smaller cost when putting the smaller relation as the outer relation!

Let |R| denotes the number of pages in R.
The entire R is scanned only one time.
The entire S is scanned |R| times.
Disk I/O Cost = |R|+|S|*|R| pages

Block Nested Loops Join

join condition R.A=S.B



```
Load B-2 pages of R from disk into the memory buffer pool
foreach block of B-2 pages of R do
  foreach page of s in S do
    // in memory
    foreach r tuple in the buffer for R
      foreach s tuple in the buffer for S
        if r.A == S.B then add <r, s> to the output buffer
        Whenever the output buffer is full, output the result
```

$Cost=|R|+|S|*\left\lceil\frac{|R|}{B-2}\right\rceil$

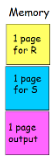
Optimal if the smaller of the two relations fit in B-2 pages of the memory

B: Available memory in pages in the database buffer pool

Indexed Nested Loops Join

join condition R.A=S.B (e.g., Recipe.fid=Food.fid)

```
foreach page of R in R do   // R: outer relation; bring in from disk one page at a time
  foreach tuple r in the buffer for R do
    use the value of r.A to search the index built on S.B
    for all matching s tuples in S
      add <r, s> to the output buffer
      output the result whenever the output buffer is full
```

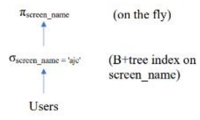


- Considered only when there is an index on either R.A or S.B
- The one with the index on the joining attribute is used as the inner relation
- Cost depends on what index is used

Single index access path plan

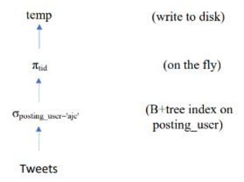
a) There is only one index relevant to the condition in the where clause and the attribute in the condition is the first attribute of the index and the attribute in the where clause is not inside a function. Hence, there is only one single index access path plan. If there were another index on screen_name, DBMS would consider the plan using that index as well.

SELECT u.screen_name FROM Users u where u.screen_name = 'ajc';



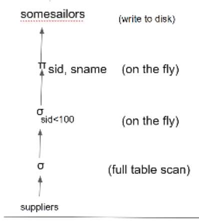
b) There is only one index relevant to the condition in the where clause and the attribute in the condition is the first attribute of the index and the attribute in the where clause is not inside a function. Hence, there is only one single index access path plan. If there were another index on posting_user, DBMS would consider the plan using that index as well.

INSERT into temp SELECT tid FROM Tweets where posting_user='ajc';



Suppliers(sid int, sname VARCHAR(30) unique not null, address VARCHAR(66), primary key(sid)).
Use the assumption as given in CP-WK11.

Group : C
insert into somesailors
select sid, sname
from suppliers
where sid<100;



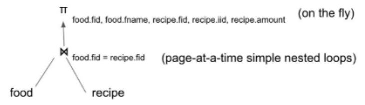
Disk I/O cost = cost of disk I/Os for doing the full table scan + cost of writing the content in somesailors to disk

#bytes per record for suppliers =4+30(*1) + 66(*1) = 100
#records/page = 20
#pages for the relation = 1 +500 = 501

Cost for writing somesailors to disk
#bytes per row for somesailors = 4+30*1=34
#records/page = floor(2000/34) = 58
#rows in somesailors = 0.1*10,000 = 1000 rows

π σ

a) Draw a query execution plan for the above query using the page-at-a-time simple-nested loops join algorithm. Let food be the outer relation and recipe be the inner relation.



π food.fid, food.fname, recipe.fid, recipe.iid, recipe.amount
(on the fly)
⋈ food.fid = recipe.fid
(page-at-a-time simple nested loops)
food
recipe

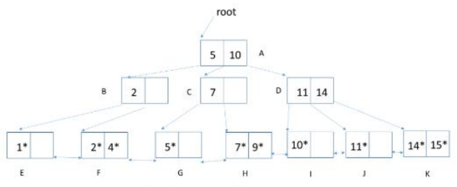


Figure 1. Dense B+Tree index on R.id

Answer the following questions. You can use the labels A, B, C, ..., K to represent the tree nodes in your answers for 1.c)-1.e).

a) What is the order of this tree? 1 is from 2 (maximum number of entries per node)/2
b) How many tuples are in the relation R? 10
c) List all the nodes that must be examined to find R.id=6. Answer: A, C,G
d) List all the nodes that must be examined to find R.id < 10. Answer: A, C, H, G, F, E
e) List all the nodes that must be examined to find R.id R.id > 2. Answer: A, B, F, G, H, I, J, K

Benefits of DBMS: reduce dev time, Data independence, Efficient Access, Data integrity, Ease of data sharing/ security but cannot model all requirements

a) Provide the absolute path of the file storing the data of the Emp table.
b) What are indexes associated with the Emp table? PRIMARY index built on the primary key attribute eid
c) Which of the following queries, a full table scan is used? (a), (c), (d) where there is no index on ename
d) Which of the following queries, some indexes are used? (b), (d) when the index on ename exists, (e), (f), (g)
e) Which of the following queries, a join algorithm is used? (g) where the index nested loops join algorithm is used.

(a)	Select * FROM Emp;
(b)	SELECT * FROM Emp WHERE eid = 101;
(c)	SELECT * FROM Emp WHERE upper(ename) = 'JOHN';
(d)	SELECT * FROM Emp WHERE ename = 'John';
	Create index enameIdx on Emp(ename); See what happen when you run queries (c) and (d) again. Then run the following statement. drop index enameIdx on emp;

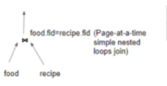
(e)	SELECT count(*) FROM Emp;
(f)	SELECT did, dname FROM Dept where did < 10;
(g)	SELECT e.eid, e.ename, w.did FROM emp e inner join works w on e.eid = w.eid;

b) Work through the page-at-a-time simple-nested loops join algorithm for the query using the above instances of food and recipe relations.

We focus on the join operation.

b) Each buffer is of the size of 1 page

1st page of the outer relation and 1st page of the inner relation



Buffer for the outer relation Food (fid, fname)

1	Pizza
2	Hummus

Buffer for the inner relation Recipe (fid, iid, amount)

1	1	50g
1	2	60g

Buffer for the output (food.fid, food.fname, recipe.fid, recipe.iid, recipe.amount)

1	Pizza	1	2	60g

The buffer for the output above is after the innermost loop is executed. Initially, the buffer was empty. This buffer becomes full when the first row of food is joined with the first row of recipe, and the content of the row is given to the next operator in the query execution plan. The above output is after the first row of food is joined with the 2nd row of recipe.

```
try {

    conn.setAutoCommit(false);
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
    // Static SQL statement uses Statement object
    Statement stmt = conn.createStatement();
    ResultSet rs;

    String tfname;
    int id=0;
    String sqlstr="select fid, fname from food where fid = (select max(fid)from food)";

    // this query has a problem that fname may not correspond to the
    // maximum fid value; however, we do not use tfname for anything
    rs = stmt.executeQuery(sqlstr);

    while (rs.next()) {
        id = rs.getInt(1);
        tffname = rs.getString(2);
    }
    rs.close();
    stmt.close();

    // use of a parameterized SQL statement which is a statement with
    // the question mark whose value is to be replaced by the
    // parameter values given by a user

    PreparedStatement inststmt =
    conn.prepareStatement("update food set fname=? where fid=?");

    String fname=JOptionPane.showInputDialog("Enter food name:");
    inststmt.setString(1, fname.toUpperCase());
    inststmt.setInt(2, id);
    inststmt.executeUpdate();
    inststmt.close();
    conn.commit();

} catch (SQLException e) {}
```

Program 2

Questions:

1. What does Program 1 do?
   Answer: Program 1 enters a new row in the food table with the fid value equal to one plus the highest fid value in the table and with the fname value equal to the application user's input string.

2. What does Program 2 do?
   Answer: Program 2 updates the fname value of the row with the highest fid value in the food table with the upper case string of the application user's input.