Database Final Exam Notes

SQL

```
1. create table employee2 (                              -- fewer columns than table employe

      fname    varchar(15) not null,

      lname    varchar(15) not null,

      ssn        char(9)        not null,

      salary   decimal(10,2),

      super_ssn char(9),

      dno      int              not null,

primary key (ssn),

foreign key (super_ssn) references employee(ssn),

foreign key (dno) references department(dnumber)

);


2. create table department2 (                -- again, fewer columns than department

    dname       varchar(15)   not null,

    dnumber   int                  not null,

    mgr_ssn    char(9)          not null,

   primary key (dnumber),

   foreign key (mgr_ssn) references employee(ssn)

);

3. insert into employee2 values ('peter', 'dordal', '123456789', 29000.01, '012345678', 55);
4. delete from employee2 where fname='peter';
5. update employee2 set salary = 1.10 * salary where salary >= 50000;
6. select e.lname from employee2 e where e.dno = 5;
7a. select e.lname from employee2 e, department2 d where e.dno = d.dnumber and d.dname="maintenance";
7b. select e.lname from employee2 e join department2 d on e.dno = d.dnumber where d.dname="maintenance"; (same result as 7a)
8. select e.lname from employee2 e where e.salary in (select e.salary from employee2 e where e.dno = 5);
9. select e.dno, count(*) from employee e GROUP BY e.dno;
10. select e.dno, sum(e.salary) from employee e GROUP BY e.dno
11. Select sum(e.salary) from employee2 e, department2 d where d.dname='Research' and d.dnumber=e.dno;
12.  Select e.lname, e.salary, 1.035 * e.salary AS newsalary from employee2;
```

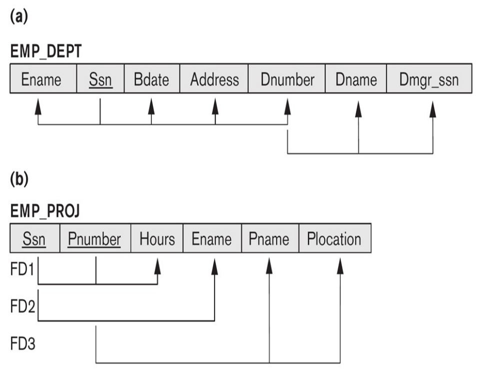
Functional Dependencies and Normalization

A functional dependency is a kind of **semantic constraint**. If X and Y are sets of attributes (column names) in a relation, a functional dependency X⟶Y means that if two records have equal values for X attributes, then they also have equal values for Y. Like key constraints, FD constraints are not based on specific sets of records. For example, in the US, we have {zipcode}⟶{city}, but we no longer have {zipcode}⟶{areacode}.

Example 1: **EMP_DEPT**
⟨Ename, <u>Ssn</u>, Bdate, Address, Dnumber, Dname, Dmgr_ssn⟩
Dependencies:
Ssn ⟶ Ename, Bdate, Address, Dnumber
Dnumber ⟶ Dname, Dmgr_ssn

Example 2: **EMP_PROJ**
⟨<u>Ssn</u>, <u>Pnumber</u>, Hours, Ename, Pname, Plocation⟩
Dependencies:
Ssn ⟶ Ename
Pnumber ⟶ Pname, Plocation
{Ssn, Pnumber} ⟶ Hours

Figure 15.3
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.



A **superkey** (or **key superset**) of a relation schema is a set of attributes S so that no two tuples of the relationship can have the same values on S. A **key** is thus a **minimal superkey**: it is a superkey with no extraneous attributes that can be removed. For example, {Ssn, Dno} is a superkey for EMPLOYEE, but Dno doesn't matter (and in fact contains little information); the key is {Ssn}.

Relations can have multiple keys, in which case each is called a **candidate key**. For example, in table DEPARTMENT, both {dnumber} and {dname} are candidate keys. For arbitrary performance-related reasons we designated one of these the **primary key**; other candidate keys are known as **secondary keys**.

A **prime attribute** is an attribute (ie column name) that belongs to *some candidate key*. A nonprime attribute is not part of any key.

A dependency X⟶A is **full** if the dependency fails for every proper subset X' of X; the dependency is **partial** if not, ie if there *is* a proper subset X' of X such that X'⟶A.

Normal Forms and Normalization

First Normal Form

First Normal Form (1NF) means that a relation has no composite attributes or multivalued attributes. Note that dealing with the multi-valued **location** attribute of DEPARTMENT meant that we had to create a new table LOCATIONS. Composite attributes were handled by making each of their components a separate attribute.

Second Normal Form

Second Normal Form (2NF) means that, if K represents the set of attributes making up the primary key, every **nonprime**attribute A (that is an attribute not a member of any key) is functionally dependent on K (ie K⟶A), but that this fails for any proper subset of K (no proper subset of K functionally determines A). Alternatively, 2NF means that for every nonprime attribute A, the dependency K⟶A is **full**.

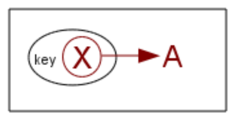
In the **EMP_PROJ** relationship, the primary key K is {Ssn, Pnumber}. 2NF fails because {Ssn}⟶Ename, and {Pnumber}⟶Pname, {Pnumber}⟶Plocation.

Third Normal Form

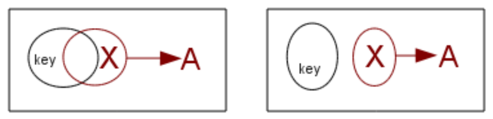
Third Normal Form (3NF) means that the relation is in 2NF and also there is no dependency X⟶A for nonprime attribute A and for attribute set X that *does not contain* a candidate key (ie X is not a superkey). In other words, if X⟶A holds for some nonprime A, then X must be a superkey. (For comparison, 2NF says that if X⟶A for nonprime A, then X cannot be a proper subset of any key, but X can still overlap with a key or be disjoint from a key.)

2NF: If K represents the set of attributes making up the primary key, every **nonprime** attribute A (that is an attribute not a member of any key) is functionally dependent on K (ie K⟶A), but that this fails for any proper subset of K (no proper subset of K functionally determines A).

3NF: 2NF + there is no dependency X⟶A for nonprime attribute A and for an attribute set X that *does not contain* a key (ie X is not a superkey).



X→A disallowed by 2NF



Dependencies X→A disallowed by 3NF

Normalization

A dependency X⟶A that violates 2NF or 3NF can be fixed by **factoring out**: we remove column A from the original table, and create a new table ⟨X,A⟩. For example, if the table ⟨K1, K2, A, B⟩ has dependency K1,A⟶B (violating 3NF) we create two new tables ⟨K1, K2, A⟩ and ⟨K1, A, B⟩. If we were factoring out A⟶B, we would create new tables ⟨K1, K2, A⟩ and ⟨A,B⟩. Both the resultant tables are *projections*of the original; in the second case, we also have to remove duplicates. Sometimes there are differences in the end result depending on which dependency we chose first for factoring out.

The relationship EMP_DEPT of EN fig 15.11 is not 3NF, because of the dependency dnumber ⟶ dname (or dnumber ⟶ dmgr_ssn). If we factor by dependency **dnumber ⟶ dname, dmgr_ssn**, we get

ED1: ⟨Ename, Ssn, Bdate, Address, Dnumber⟩, ED2: ⟨Dnumber, Dname, Dmgr_ssn⟩

Boyce-Codd Normal Form

BCNF requires that whenever there is a nontrivial functional dependency X⟶A, then X is a superkey. It differs from 3NF in that 3NF requires *either* that X be a superkey *or* that A be prime (a member of some key). To put it another way, BCNF bans all nontrivial nonsuperkey dependencies X⟶A; 3NF makes an exception if A is prime.

As for 3NF, we can use factoring to put a set of tables into BCNF. However, there is now a serious problem: by factoring out a *prime* attribute A, we can destroy an existing key constraint! This is undesireable.

The canonical example of a relation in 3NF but not BCNF is ⟨A, B, C⟩ where we also have C⟶B. Factoring as above leads to ⟨A, C⟩ and ⟨C, B⟩. We have lost the key A,B! However, this isn't quite all it appears, because from C⟶B we can conclude A,C⟶B, and thus that A,C is also a key, and *might* be a better choice of key than A,B.

Generally, it is good practice to normalize to 3NF, but it is often not possible to achieve BCNF. Sometimes, in fact, 3NF is too inefficient, and we re-consolidate for the sake of efficiency two tables factored apart in the 3NF process.

File organizations

The simplest file is the **heap** file, in which records are stored in order of addition. Insertion is efficient; search takes linear time. Deletion is also slow, so that sometimes we just mark space for deletion.

Another format is to keep the records ordered (sorted) by some field, the **ordering field**. This is not necessarily a key; for example, we could keep file Employee ordered by Dno. If the ordering field is a key, we call it the **ordering key**. Ordered access is now fast, and search takes log(N) time (where N is the length of the file *in blocks* and we are counting only block accesses).

Insertion and deletion are expensive. We can improve insertion by keeping some unused space at the end of each block for new records (or the moved-back other records of the block). We can improve deletion by leaving space at the end (or, sometimes, right in the middle). Another approach is to maintain a **transaction file**: a sequential file consisting of all additions and deletions. Periodically we update the master file with all the transactions, in one pass.

Hashing as file organization

We have a hash function h that applies to the key values, h = hash(key). For disk files, we typically use full blocks as buckets. However, these will often be larger than needed. As a result, it pays to consider hash functions that do not generate too many different values; a common approach is to consider hash(key) mod N, for a smallish N (sometimes though not universally a prime number).

Given a record, we will compute h = hash(key). We also provide a single-block map ⟨hash,block⟩ of hash values to block addresses (in effect corresponding to hashtable[]). Note that in Fig 17.10, Bucket 1 and Bucket 2 *share* an overflow bucket; we also can manipulate the ⟨hash,block⟩ structure so that two buckets share a block. When a single bucket approaches two blocks, it can be given its own overflow block.

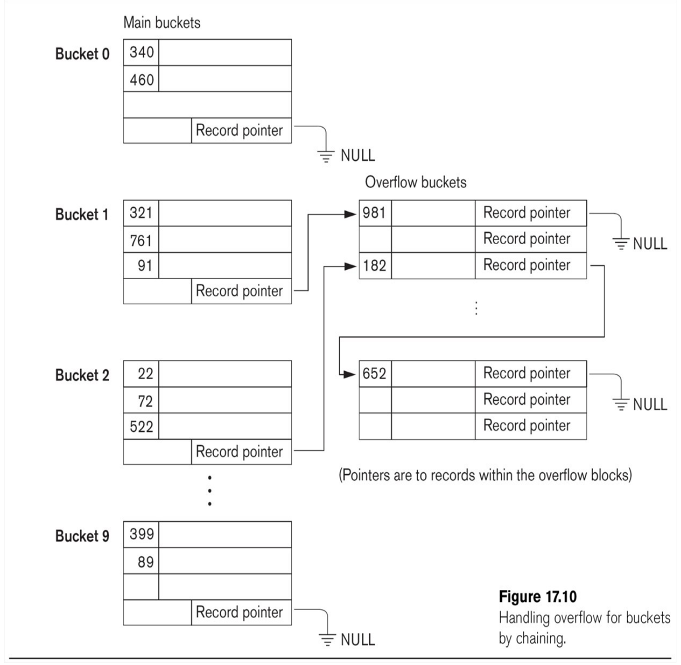


Figure 17.10
Handling overflow for buckets by chaining.

Extendible Hashing

This technique manages buckets more efficiently. We hash on the first *d* bits of the hash values; d is called the **global depth**. We keep a directory of all 2^d possible values for these d bits, each with a pointer to a bucket block. Sometimes, two neighboring buckets are **consolidated** in the directory; for example, if d=3, hash prefix values 010 and 011 might point to the same block. That block thus has a reduced **local depth** of 2.

As we fill blocks, we need to create new ones. If a block with a reduced local depth overflows, we split it into two blocks with a greater depth (still ≤ d). If a block with local depth d overflows, we need to make some global changes: we increment d by 1, double the directory size, and double up all existing blocks except for the one causing the overflow.

Ch 18: indexing

It is common for databases to provide **indexes** for files. An index can be on either a key field or a non-key field; in the latter case it is called a **clustering index**. The index can either be on a field by which the file is sorted or not. An index can have an entry for *every* record, in which case it is called **dense**; if not, it is called **sparse**. An index on a nonkey field is always considered sparse, since if every record had a unique value for the field then it would in fact be a key after all.

A file can have multiple indexes, but the file itself can be structured only for one index. We'll start with that case. The simplest file structuring for the purpose of indexing is simply to keep the file sorted on the attribute being indexed; this allows binary search. For a while, we will also keep restrict attention to **single-level indexes**.

A **primary index** is an index on the primary key of a **sorted** file (note that an index on the primary key, if the file is not maintained as sorted on that primary key, is thus *not* a "primary index"!). The index consists of an ordered list of pairs ⟨k,p⟩, where k is the first key value to appear in the block pointed to by p (this first record of each block is sometimes called the **anchor record**). To find a value k in the file, we find consecutive ⟨k1,p⟩ and ⟨k2,p+1⟩ where k1≤k<k2; in that case, the record with key k must be on block p. This is an example of a **sparse index**. A primary index is usually much smaller than the file itself.

Example 1 on EN6 p 635: 30,000 records, 10 per block, for 3000 blocks. Direct binary search takes 12 block accesses. The index entries are 9+6 bytes long, so 1024/15 = 68 fit per 1024-byte block. The index has 3000/68 = 45 blocks; binary search requires 6 block accesses, plus one more for the actual data block itself.

Clustering index

We can also imagine the file is ordered on a nonkey field (think Employee.dno). In this case we create a **clustering index**. The index structure is the same as before, except now the block pointer points to the first block that contains any records with that value. Clustering indexes are of necessity sparse.

Secondary Indexes

Now suppose we want to create an index for Employee by (fname, lname), assumed for the moment to be a secondary **key**. The record file itself is ordered by Ssn. An index on a **secondary key** will necessarily be dense, as the file won't be ordered by the secondary key; we cannot use block anchors. A common arrangement is simply to have the index list ⟨key,block⟩ pairs for every key value appearing; if there are N records in the file then there will be N in the index and the only savings is that the index records are smaller. If B is the number of blocks in the original file, and BI is the number of blocks in the index, then BI ≤B, but not by much, and log(BI) ≃ log(B), the search time. But note that unindexed search is *linear* now, because the file is not ordered on the secondary key.

Example 2, EN6, p 640: 30,000 records, 10 per block. Without an index, searching takes 1500 blocks on average. Blocks in the index hold 68 records, as before, so the index needs 30,000/68 = 442 blocks; $\log_2(442) \simeq 9$.

Secondary indexes can also be created on **nonkey** fields. One common option is for each index entry to point to *blocks* of record pointers.

Hashing in Indexes

Hashing can be used to create a form of index, even if we do not structure the file that way. We can use hashing with equality comparisons, but not order comparisons, which is to say hashing can help us find a record with ssn=123456789, but not records with salary between 40000 and 50000.

ISAM Indexes

Perhaps our primary sorted index grows so large that we'd like an index for *it*. At that point we're creating a multi-level index. To create the ISAM index, we start with the primary index, with an index entry for the anchor record of each block, or a secondary index, with an entry for each record. Call this the **base level**, or **first level**, of the index. We now create a **second level** index containing an index entry for the anchor record of each block of the first-level index. This is called an **indexed sequential file**, or an **ISAM** file. This technique works as well on secondary keys, except that the *first* level is now much larger.

B-trees (Bayer trees)

Given an odd integer **order**, p, a B-tree of order p is an ordered tree in which each node has at most **p** tree pointers, and **p-1** key values. In addition, all but the top node has at *least* **(p-1)/2** key values.

To add a new value we use the "push-up" algorithm: we add the new value to the appropriate leaf block. If there is room, we are done. If not, this means the leaf block now has **p** key values. We **split** the leaf block into two of size **(p-1)/2** and push up the middle value to the parent block. The parent block may now also have to be split.

Query processing

To implement record **selection** (SQL **where** clauses), we can use

- **Linear search**: we read each disk block once.
- **Binary search**: this is an option if the selection condition is an equality test on a key attribute, and the file is ordered by that attribute.
- **Primary index**: same requirement as above, plus we need an index
- **Hash search**: equality comparison on any attribute for which we have a hash index (need not be a key attribute)
- **Primary index and ordered comparison**: use the index to find all departments with dnumber > 5
- **B-tree index**

To implement **joins**, we have:

Book example, EN6 p 690-693: Suppose we have 6,000 employee records in 2,000 blocks, and 50 departments in 10 blocks. We have indexes on both tables. The employee.ssn index takes 4 accesses, and the dept.dnumber index takes 2.

1. **Nested-loop join**: this is where we think of a for-loop:

```
for (e in employee) {
   for (d in department) {
      if (e.dno = d.dnumber) print(e,d);
   }
}
```

This is quadratic. If we go through the employee table record by record, that amounts to 2,000 block accesses. For each employee *record* the loop above would go through all 10 blocks of departments; that's 6,000 × 10 = 60,000 blocks. Doing it the other way, we go through 10 blocks of departments, and, for each department record, we search 2,000 blocks of employees for 50×2,000 = 100,000 block accesses.

However, note that we can also do this join **block-by-block** on both files. Done this way, the number of block accesses is 2,000 × 10 = 20,000 blocks.

Performance improves rapidly if we can keep the smaller table entirely in memory: we then need only 2,010 block accesses!

2. **Index join**: if we have an index on one of the attributes, we can use it:

```
for (e in employee) {
   d = lookup(department, e.ssn);
   if (d != null) print(e,d);
}
```

Note that any index will do, but that this *may* involve retrieving several disk blocks for each e and will almost certainly involve retrieving at least one disk block (from department) for every e in employee. It may or may not be better than Method 1. Consider the first query. Suppose we have a primary index on department.dno that allows us to retrieve a given department in 2 accesses. Then we go through 6,000 employees and retrieve the department of each; that's 6,000×2 = 12,000 block accesses. Now consider the second query, and suppose we can find a given employee in 4 accesses. Then we go through 50 × 4 = 200 block accesses (for every department d, we look up d.mgr_ssn in table employee).

3. **Sort-merge join**: we sort both files on the attribute in question, and then do a join-merge. This takes a single linear pass, of size the number of blocks in the two files put together file. This is most efficient if the files are *already* sorted, but note that it's still faster than 1 (and possibly faster than 2) if we have to sort the files. Assume both tables are sorted by their primary key, and assume we can sort in with N log(N) block accesses, where N is the number of blocks in the file. Then query 1 requires us to sort table employee in time 2,000×11 = 22,000; the actual merge time is much smaller. Query 2 requires us to sort table department in time 10×4 = 40; the merge then takes ~2,000 blocks for the employee table.

4. **Partition-hash join**: Let the relations (record sets) be R and S. We partition both files into Ri = {r in R | hash(r) = i}. Now we note that the join R ⋈ S is simply the disjoint union of the Ri ⋈ Si. In most cases, either Ri or Si will be small enough to fit in memory.

The **join selection factor** is the fraction of records that will participate in the join. In query 2 above,

```
select * from department d, employee e where d.mgr_ssn = e.ssn
```

all departments will be involved in the join, but *almost no employees*. So we'd rather go through the departments, looking up managing employees, rather than the other way around.

In the most common case of joins, the join field is a foreign key in one file and a primary key in the other. Suppose we keep all files sorted by their primary key. Then for any join of this type, we can traverse the primary-key file block by block; for each primary-key value we need to do a lookup of the FK attribute in the other file. This would be method 2 above; note that we're making no use of the primary index.

Transactions

A transaction is a set of operations, which we can idealize as read(X), write(X), which at the end we either **commit** (save) or **rollback/abort** (discard).

Abstract transactions: operations are read(X) and write(X), for X a field/record/block/table.
two example transactions.

T1: read(X), write(X), read(Y), write(Y)
T2: read(X), write(X)

Ideally, we execute the transactions one at a time, with no overlap. In practice, because of the considerable I/O-wait for disk retrieval, we cannot afford that. Transactions will **overlap**. We need to figure out how to arrange for this to happen **safely**. Two problems:

- Lost Update (T1 writes X, T2 then overwrites X)
- Dirty Read w Abort (T1 writes X, T2 reads X, T1 aborts)

The ACID test

This is commonly described as the following set of features.

- **Atomicity**: either all steps of a transaction are performed, or none are. We either commit the entire thing or rollback completely.
- **Consistency** preservation: the individual transaction shoud preserve database consistency and database invariants
- **Isolation**: transactions should appear to be executing in isolation (ideally they appear to be executing serially)
- **Durability**: committed transactions survive DB crashes

Transaction schedules

Transaction schedules are lists of read/write operations of each transaction, plus commit/abort. The operations of each individual transaction must be in their original order. Example:

T1	T2

read(X)
X = X-N	

	read(X)
	X = X+M

write(X)
read(Y)	

	write(X)

Y = Y+N
write(Y)	

Schedule: read1(X), read2(X), write1(X), read1(Y), write2(X), write1(Y), commit1, commit2

Two operations is a schedule **conflict** (where the word is to be taken as indicating *potential* conflict) if:

1. They belong to different transactions
2. They access the same data item (eg X)
3. At least one of the operations is a write()

For example, in the schedule above, r1(X) and w2(X) conflict. So do r2(X) and w1(X). These do **not** conflict: r1(X) and r2(X), r1(X) and w1(X).

Conflicting operations are those where interchanging the order can result in a different outcome. A conflict is **read-write** if it involves a read(X) in one transaction and write(X) in the other, and **write-write** if it involves two transactions each doing write(X). A **complete schedule** is a total order of all the operations, incuding abort/commits. (The book allows some partial ordering of irrelevant parts). Given a schedule S, the **committed projection** C(S) is those operations of S that are part of transactions that have committed. This is sometimes useful in analysing schedules when transactions are continuously being added.

We can build a **transaction precedence graph** based on schedule conflict, given a schedule S. Let us say Ti ⟶ Tj if Ti and Tj conflict as above, and one of the following occurs:

1. Ti executes write(X) and later Tj executes read(X)
2. Ti executes read(X) and later Tj executes write(X)
3. Ti executes write(X) and later Tj executes write(X)

Recoverable and nonrecoverable schedules

Given two transactions T and T', we say T **reads from** T', or T **depends on** T', if T' does a write(X) and then later T does a read(X) (this is Case 1 of T'⟶T above) Thus, the outcome of T may depend on the (at least in part) earlier T'.

A schedule is **recoverable** if no transaction T can commit until all T' where T reads from T' have already committed. Otherwise a schedule is **nonrecoverable**. If T reads from T', then if T' aborts, T must abort too. So T waits to commit until T' commits. Given a recoverable schedule, **we never have to abort a committed transaction**. This is, in practice, an essential rule.

Consider the following modification of the schedule above:
Schedule2: read1(X), read2(X), write1(X), read1(Y), write2(X), commite2, write1(Y), commit1.
This is recoverable even though there is a risk of **lost update**. But now consider
Schedule3: r1(X), w1(X), r2(X), r1(Y), w2(X), c2, abort1 **non**recoverable: T2 reads X from T1 but T2 commits first
Schedule4: r1(X), w1(X), r2(X), r1(Y), w2(X), w1(Y), c1, c2; recoverable because T2 waited to commit until T1 did

Though recoverability implies no committed transaction needs to be rolled back, we still may have **cascading rollback**. A schedule is said to **avoid cascading rollback** if every transaction T only reads items that were earlier written by **committed** transactions. In this case, an abort of some other transaction will not cause T to abort. We can achieve this in schedules 3 and 4 by delaying read2(X) until after commit1.

Serializability

The ideal situation with transactions is that they are executed serially, but that is too inefficient. Instead, we look for schedules that are in some sense *equivalent* to a serial schedule. A schedule is **serializable** if it is **equivalent** to a serial schedule. For equivalence, we usually use **conflict equivalence**. Two schedules are **conflict-equivalent** if the order of any two **conflicting** operations is the same in both schedules.

Conflict-serializability algorithm

Build the transaction precedence (directed) graph above. If the graph has no cycles, it is conflict-serializable. Do a topological sort (ie find a total order of the nodes consistent with the Ti⟶Tj ordering); the original schedule is conflict-equivalent to the schedule of executing the transactions in that order.

The actual algorithm is as follows: first, find all Ti that do not have any predecessors, ie Tk with Tk⟶Ti. Do these first, and remove all Ti⟶Tj links. Now repeat, until there are no more nodes. If at some stage there are no Ti without predecessors, then there must be a cycle.