11/08/17 04:35:17 D:\Google Drive Sync\Programming\UNI\dbs\summary.notes

```
--------------------------------------------------------------------------------
    Database Systems (i.e. the worst core subject of the CS undergrad)
--------------------------------------------------------------------------------

Data
- Known facts stored and recorded
- Complex objects such as text, numbers etc.
- Raw information
- e.g. a name or age of someone

Information
- Data presented in context (can be summarised data)
- Data that has been processed increasing the users knowledge.
- e.g. "Matt de Bono started teaching Object Oriented Software Development in
       2017"
// and really should've taught this class too

Data #is known and available
Information #is processed and more useful

!What is a Database?:
- A large, integrated, structured collection of data
- Usually intended to model some real-world enterprise

!What is a Database System?:
- A <Database Management System> is a software system designed to
        - Store
        - Manage
        - Facilitate access to databases


Manipulate data with a query language (e.g. SQL)

#Types of Query Languages:

Data definition language (DDL):
- Defines and sets up the database

Data manipulation language (DML):
- Maintains and uses the database

Data control language (DCL):
- To control access to the database

--- Problems you run into without Databases ---

# Program data dependence
- If the <file structure changes>, so does the program
- Programs "knows" too much about low-level data structure (access control)

# Redundant information
- Wasteful, <inefficient>, loss of data integrity
- Loss of metadata integrity
    - e.g. <same name different data>
```

```
# Limited Data Sharing
- Data is <tied to application> --> <hard/slow to create adhoc reports>

# Lengthly development times:
- Too much dependence on the application means the application has to do a lot
  of low level data management, figure out file format each time

# Excessive program maintenance
- Up to 80% of development time in traditional file based organisations is for
  maintenance

// and also no way to use the greatest programming language known to mankind:
// spicy query language

--- Good things about Databases ---

# Data Independence
- Separation of data and program, application logic
- Central data repository, central management

# Minimal Data Redundancy
- Redundancy can be controlled via <normalisation>

# Normalisation
- Breaking down a relationship between two relations into a set of smaller
  relations

# Improved data sharing
- Data is shared
- External users can access
- Multiple views of data

# Reduced program maintenance
- Data structure can change without application changing

# Increased productivity of application development
- Data <already collected> and <structures already known>
- DBMS provides many tools to help access and manipulate data

# Enforcements of standards
- Centralised data management
- Documented policy for data management
- Data definition and dictionary (metadata)

# Improved data quality
- Constraints <built into the database>

// and a dead end career in a 9-5 enterprise job that you hate

So now with databases we can manage data in a structured way

Using the <relational model>
- <Rows> and <Columns> form <relations>
- <Keys> and <Foreign Keys> to link <relations>
// Basically just tables and headings like in spreadsheets
```

!Data Model:
- Collection of concepts for describing data

!Schema:
- Description of a particular collection of data, using a given data model

!Relational model of data:
- Most widely used model
- Relations have schemas which describes columns and fields

!Views:
- Describe how users see the data

!Conceptual Schema:
- A conceptual schema defines the logical structure

!Physical Schema:
- Describes the files and indexes used.

---

Database Application Development Lifecycle

---

Literally that thing from High School
For this subject, we will only focus on <design>, not any of the maintenance / reevaluation stuff

# Database Design
- Conceptual
- Logical
- Physical

!Conceptual Design:
- Construction of a model of the data used in the database
- Independent of all physical considerations
- ER/EER diagrams

!Logical Design:
- Construction of the model based on the conceptual design
- Based on conceptual models
- Independent of a specific database and other physical considerations
- Decisions like what type of database should be used.
- Finding constraints
    - What datatypes to use?

!Physical Design:
- Actually physically making the database
- Description of the implementation of the logical design for a specific DBMS
- Describes:
    - Basic relations
    - File organisations
    - indexes
    - Integrity constraints
    - Security measures

---

Conceptual Design

---

# Guide to Conceptual Design
- What are the <entities> and <relationships> in the enterprise?
- what <information> about these entities and relationships should we store in the database?
- What are the <integrity constraints> that hold?
- A database "schema" in the ER model can be represented as a ER diagram
- Can map an ER diagram into a relational schema

# ER Model Basics
**!Entity:**
- Real world object distinguishable from other objects
- An entity is described using a set of <attributes>

**!Entity Set:**
- A collection of similar entities, e.g. <all employees>
- All entities in an entity set have the same set of attributes
- Each entity set has a <key> (underlined)
- Each attribute has a <domain>

**!Relationship**
- Association among two or more entities (e.g. Fred works in Pharmacy dept)

**!Relationship Set**
- Collection of similar relationships

Same entity set can participate in different relationship sets, or in different roles in the same set.

# Design Choices:
- Should a concept be modeled as an <entity or an attribute>?
- Should a concept be modeled as an <entity or a relationship>?
- Identifying relationships: <Binary or ternary>?

# Constraints in the ER Mode
- A lot of data semantacs can and should be captured

--- Summary of Conceptual Design ---
- Yields a high level description of data to be stored

# ER model popular for conceptual design
- Constructs are expressive, and close to the way people think about their applications
- Originally proposed by Peter Chen, 1976

# Basic Constructs:
- <Entities>, <Relationships> and <Attributes>
- <Weak Entities>

---

Key Constraints

---

!Participation Constraint:
- The participation of Departments in Manages is said to be <total> (vs partial)
- When a attribute in one relation <must appear in another relation's rows>.
e.g. A department relation with the attribute manager id also must appear
     in the manager relation.
- Basically means "at least one"

!Weak Entity:
- Can be identified uniquely <only by considering the primary key of another>
  entity
- Owner entity set and weak entity set must participate in one-to-many
  relationship set
- Weak entity set must have total participation in this identifying relationship

Weak entities only have a "partial key" (dashed underline)

Entity vs. Attribute:
- Should we model an address of an Employee to be an attribute or an entity?
- <Depends upon how we want to use address information> and the semantics of the
  data
    - If we have <several addresses per employee>, address must be an entity
    - If the <structure is important> (e.g. city, street, etc.), address should
      be modeled as an entity

---

ER design is <subjective>, there are many ways to model a given scenario

<Relational Database> - A set of <relations>

Relation:
- Made up of two parts, the <Schema> and the <Instance>

Cardinality:
- Rows

Degree / Arity:
- Fields

Can think of a relation as a set of rows or tuples
- i.e. all rows are distinct, no order among rows

Basics of SQL:
[sql]
CREATE TABLE <name> (<field> <domain>, ...)

INSERT INTO <name> (<field names>)
    VALUES  (<field values>)

DELETE FROM <name>
    WHERE <condition>

UPDATE <name>
    SET <field name> = <value>
    WHERE <condition>

```
SELECT <fields>
    FROM <name>
    WHERE <condition>
[/sql]

!Primary Key:
- A set of fields is a <super key> if
    - No two distinct tuples can have the same values in all key fields
    - Super key is like when you use <multiple attributes> to make the Keys

- A set of fields is a key for a relation if:
    - It is a super key
    - No subset of the fields is a superkey

What if there is >1 key for a relation?
- One of the keys chosen to be the <primary key>, other keys are called
  <candidate keys>

It's *redundant* to have many <candidate keys unique> and have <one primary key>
Better just to have a <composite primary key>

!Foreign Key:
- Set of fields in one relation that is <used to refer to a tuple> in another
  relation
- If all foreign key constraints are enforced, <referential integrity> is
  achieved

Translating Conceptual to Logical:
Add constraints and keys to ER
- Underline                = PK    (Primary Key)
- Italic and Underline     = FK    (Foreign Key)
- Underline and Bold       = PFK   (Primary Foreign Key)

From Logical to Physical Design:
Just make it in SQL lol

Logical to Physical Example:
Contracts (supplier_id, part_id, department_id) <- Imagine these are bold
                                                   and underlined (PFK)
---
translates to
---
[sql]
CREATE TABLE Contracts (
    supplier_id INTEGER,
    part_id INTEGER,
    department_id INTEGER,
    PRIMARY KEY (supplier_id, part_id, department_id),
    FOREIGN KEY (supplier_id) REFERENCES Suppliers,
    FOREIGN KEY (part_id) REFERENCES Parts,
    FOREIGN KEY (department_id) REFERENCES Departments)
[end]

Summary of the Relational Model:
- Tabular representation of data
```

- Simple
- Integrity constraints

// Skip lecture 5 because there's no way they could test us on how to workbench
// Note: I am not responsible for any damages to H1 dreams caused by this
// statement

// what is this subject structure

# Unary and Ternary Relationships

One-to-One and One-to-Many require <one foreign key> in the corresponding
relation

Many-to-Many you should generate an <associative entity> and <put two foreign>
<keys in the associative entity> (so you end up with two one-to-many relations)

# EER (Enhanced Entity Relation) Design

Basically most of database design depends on <how much you want to abstract> the
data, this whole thing is subjective and dependent on context.
// holy wow this subject is a meme

e.g.
For a database for a company that sells vehicles and requires a bunch of
different attributes, you could:
- separate it into the subtypes of vehicle (kinda like in OOP)
- or have it all at once in one relation (because you're a mad lad)

You'll need to explain why you made your choice!

For the first solution you could say <better query speeds> since less has to be
loaded into memory per relation.
Or for the second solution you could say <better space storage>.

Usually speed vs size or some weird query that you have to fulfil

Supertypes and Subtypes:
- Work the <same way as they do in OOP>
- <Supertype acts> kindof <like an super class>
- Subtypes inherit attributes (the same way subtypes extend supertype)
- Can be <disjointed> (only one subtype per instance) or <overlapping>

Crow's feet notation:
box with (d) - OR (d) =
Disjointed with one dash means that the instance can have <no subtype> if
necessary

Chen's:
double line to circle containing D or O

Note that links go from one subtype to a super type means that there is <only a>
<link between that specific subtype> (it is possible to have a subtype that is
    not related to a supertype).

// time for RELATIONAL ALGEBRA

```
--------------------------------------------------------------------------------
                            Relational Algebra
--------------------------------------------------------------------------------

Query Language /= Programming Language

Relational Algebra:
- More operational
- Useful for representing execution plans

Relational Calculus:
- Describes what you want instead of how to compute it

------------------------------------------------------------------------
!Selection (σ):! // Unicode isn't playing nice so I'll use o
- Selects a subset of <rows> from relation

°rating>8(s2)  // gets everything in s2 with rating above 8


!Projection (π):
- Retains only wanted <columns> from relation

You use these by raising them to supertype ---
for example, if I wanted to extract "sname" and "rating"
from a relation called <s2>:

// for some reason I can't do superscript π and σ. :(

pisname,rating(s2)  // gets sname and rating from s2

or if I wanted to get age:

piage(s2)           // gets age from s2

# You can combine operations using brackets:

pisname,rating(°rating>8(s2))

If you can, try to be more specific with which table you are projecting from
because then it is more efficient (less indexes to search)

------------------------------------------------------------------------
!Set-difference (-):
- Tuples in r1, but not in r2 (as in the difference between the two relations)

To get the intersection: r1 - r2

!Union (U):
- Tuples in both r1 or r2

To get the union: r1 U r2

!Intersection (∩):
- Tuples that exist in both r1 and r2

```

To get the intersect: r1 ∩ r2

---

**!Cross-Product (×):**

- Allows you to combine two relations

S1 × R1: each row of S1 <paired with> each row of R1
(This is basically getting all permutations of every row)
[sql]

```
SELECT * FROM table1, table2;
```

[end]

***Joining***

**!Natural Join (⋈):**

- Most common join, if one field (key) appears in both relations, then protect all common attributes and copy each of the unique ones:

Example:

| TableA | | TableB | |
|---|---|---|---|
| Column1 | Column2 | Column1 | Column3 |
| 1 | 2 | 1 | 3 |

TableA⋈TableB =

[sql]

```
SELECT * FROM TableA NATURAL JOIN TableB
```

[end]

| column1 | column2 | column3 |
|---|---|---|
| 1 | 2 | 3 |

**!Inner Join (or condition join):**

- Similar to natural join but puts the other row next to the row with the same specified condition

Example:

| TableA | | TableB | |
|---|---|---|---|
| Column1 | Column2 | Column1 | Column3 |
| 1 | 2 | 1 | 3 |

$TableA \bowtie_e TableB = \sigma_e(R \times S)$ // Where condition e is true

[sql]

```
SELECT * FROM TableA INNER JOIN TableB USING (Column1)
SELECT * FROM TableA INNER JOIN TableB ON TableA.Column1 = TableB.Column1
```

[end]

| a.column1 | a.column2 | b.column1 | b.column3 |
|---|---|---|---|
| 1 | 2 | 1 | 3 |

**!Outer Join:**

- Also works like inner join, but in cases where values do not exist for the same key in one of the tables,

You can also rename things by using an arrow:
TableA(1 -> a1)
or by using $\rho_x(E)$ <- where the Expression E is saved with the name x

---

## What is SQL?

---

SQL or SEQUEL is a query language used in relational database

The four basic commands are Create, Select, Update and Delete

DDL:
To set up/define database
- CREATE
- ALTER
- DROP

DML:
To maintain and use database
- SELECT
- INSERT
- DELETE
- UPDATE

DCL:
To control access to the database
- GRANT
- REVOKE

# Creating a table
If I wanted to create the relation Customer with the following attributes, CustomerID, CustFirstName, CustMiddleName, CustLastName, BusinessName, CustType

You would do the following:
[sql]

```sql
CREATE TABLE Customer (
    CustomerID      smallint                    auto_increment,
    CustFirstName   varchar(100),
    CustMiddleName  varchar(100),
    CustLastName    varchar(100)                NOT NULL,
    BusinessName    varchar(200),
    CustType        enum('Personal', 'Company') NOT NULL,
    PRIMARY KEY (CustomerID)
);
```

[end]

This specifies that CustomerLastName and CustType cannot be null - they are required in order to enter that row into the database.

CustomerID is simply a automatically incremented integer. It is also the primary key

```sql

CREATE TABLE Account (
    AccountID           smallint        auto_increment,
    AccountName         varchar(100)    NOT NULL,
    OutstandingBalance  DECIMAL(10,2)   NOT NULL,
    CustomerID          smallint        NOT NULL,
    PRIMARY KEY (AccountID),
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
            ON DELETE RESTRICT
            ON UPDATE CASCADE
);
[end]
```

In this case the <ON DELETE RESTRICT> and <ON UPDATE CASCADE> are used in reference to any foreign keys.
- If there is an attempt made to delete in the parent table, it fails.
- If there is an update to the parent table then <the logical pointer in the> <foreign key is also updated.>

# Aggregate Functions:
You can use these on a set of values in a column.
- AVG()
- MIN()
- MAX()
- COUNT()
- SUM()

All of these except count ignore null values.

e.g. SELECT COUNT(CustomerID) FROM Customer; //How many customers do we have

You can next things in SQL

Here's some comparison operators:
- IN / NOT IN
- ANY (true if any value returned meets condition)
- ALL (true of all values returned meet condition)

**!Views:**
A view is a relation that is <not in the conceptual/logical models> but is <made available to the user> as a virtual relation is called a #view.

You can create views in the same way as you use relations:
```sql

CREATE VIEW EmpPay AS
SELECT  Employee.ID, Employee.Name, DateHired,
        EmployeeType, HourlyRate AS Pay
        FROM Employee INNER JOIN Hourly
        ON Employee.ID = Hourly.ID
UNION
SELECT  Employee.ID, Employee.Name, DateHired,
        EmployeeType, AnnualSalary AS Pay
        FROM Employee INNER JOIN Salaried
        ON Employee.ID = Salaried.ID
```

```
UNION
SELECT  Employee.ID, Employee.Name, DateHired,
        EmployeeType, BillingRate AS Pay
        FROM Employee INNER JOIN Consultant
        ON Employee.ID = Consultant.ID;

[end]
```

You can actually insert values into views, but they won't be recorded in any relation

```
// Go do some actual queries - reading these shitty notes doesn't really lend
// itself to the language since it's pretty interactive.
// W3 has some good tutorials.

// also this subject is by far the worst core one I've done so far
// We're about at lecture 9, 15 lectures to go

// the wild ride continues
```

---

## Storage and Indexing

---

**!File:**
A collection of <pages>, each containing a <collection of records>

A DBMS must support:
- Inserting/Deleting/Modifying records
- Reading a specified record
- Scanning all records

There are 3 main file organisations:

**!Heap Files:**
- <No order> among records
- Suitable when <all records> are <accessed at the same rate>, or you want all to <retrieve all records>

This is the simplest data structure. As the file grows and shrinks, <disk pages> are <allocated> and <de-allocated>.
#This data structure is the fastest for inserts.

In an implementation using lists, each page contains <data plus two pointers> to <previous and next page>, just like <a linked list>.

The <header contains points to> the "head" (first full page) and "tail" (empty)

**!Sorted Files:**
- Records are <sorted by a certain condition>
- Best when some <records are required in a certain order>, or for retrieving a <range of records>

Also works like heap files, i.e. a linked list, however the pages and records <are ordered>.
This is faster for <searching for a specific record> because you can use

<binary search>

---

**!Comparison of Heap File vs Sorted File:**

The <performance> of databases are normally <measued in Disk I/O's>

Where B == number of data pages:

| | Heap File | Sorted File |
|---|---|---|
| Scan All | B | B |
| 1 Eq Search | 0.5B (avg) | $\log_2 B$ |
| Range Search | B | ($\log_2 B$) + (matching pages) |
| Insert | 2 | ($\log_2 B$) + 2*(B/2) // read to find pos, write |
| Delete | 0.5B + 1 | ($\log_2 B$) + 2*(B/2) |

TLDR: Heap File Faster to Insert, Sorted File Faster to Search

---

Sometimes we want to retrieve records by specifying values in one or more fields
e.g.
- Find all students in the "CIS department"
- Find all students with a gpa > 3

**!Index File Organizations:**
- "Special" Data structure

An <index> on a file speeds selections on the <search key fields> for the index
- A data structure built on top of data pages used for efficient search
- <Any subset> of the fields of a relation <can be the search key> for an index
    on the relation
- <Search key> is <not the same as key>

The <index basically stores a bunch of pointers> to a <range of data entries> -
which themselves <link to the data record itself>

There are a few different types of Indexes:
- Clustered vs Unclustered
- Primary vs Secondary
- Single Key vs Composite
- Indexing Technique
    - Tree-based, hash-based, other

Index Organisation Alternatives:
// this isn't explained so well
You can choose whether the index is linked to the record or the data itself
- whether you want variable record sizes, or just one bit of data per index
(this is the alternative 1, 2, 3 thing - alt 2 was used in assignment 3)

A good way to think of this is like keys to hash tables;

Alt 1:
- For an index, contain the actual data record

Alt 2:
- For an index, point to one record

Alt 3:
- For an index, point to a list of records

You can have <multiple indexes per file>
- e.g. B+ Tree on age, Hash index on name, etc...

**!Clustered vs Unclustered:**
- If the order of <data records> is the same as the order of <index data entries> then the index is called <clustered index>. Otherwise it is <unclustered>
- A file can have a clustered ondex on at most <one search key>
- Cost of retrieving data records through index varies greatly based on whether the index is clustered (cheaper for clustered)

**!Primary vs Secondary:**
- Primary: index key includes the file's primary key
- Secondary: has any other index

Primary index never contains duplicates (self explanatory since we're using primary key)
Secondary index may contain duplicates.

**!Hash-based Index vs Tree-based index:**
Hash-Based Index:
- Good for <equality selections>
    - Work similar to an extendible hash table with buckets
    - Can't range search cause it would just spaghetti

Tree-Based Index:
- Good for <range> selections
    - Hierarchical structure (tree) directs searches
    - Leaves contain data entries sorted by search key value
    - B+ tree: all root -> leaf paths have equal height (height)
    - So far we have seen tree-based indexes

Whats Examinable?
- Describe the alternatives (1, 2, 3)
- What is an index, when do we use them?
- Index classification

---

## Query Processing (Selections & Projections)

---

Some queries are < e x p e n s i v e >
So we need to actually think about what we're doing

### Selections

**!Simple Selections:**
Of the form
[sql]
```sql
SELECT * FROM Reserves R WHERE R.name < "C%"
```
[end]

Size of Result:

size of R * reduction factor

#Reduction Factor or "Selectivity"
is how much you reduce the relation as defined in your condition.

//note: Reserves is of size 1000 records

With no index, unsorted:
- Must essentially scan the whole relation
- Cost is number of pages in R.
e.g. Reserves = 1000 I/O

With no index, sorted:
- Cost of binary search + number of pages containing results
- For reserves = 10 I/O + [selectivity * pages]

With an index:
- Use index to find qualifying data entries,
- Then retrieve the corresponding data records
Note: Hash index can only be used on equality

Cost Factors:
1. find qualifying data entries
    (typically small, 2-4 I/O with B+ Tree, or 2.2 with hash index)
2. retrieve records

<Clustering helps a lot> with the <retrieving process>.


The approach to selecting records:
1. Find the cheapest path:
- an index or file scan with the fewest estimated page I/O

2. Retrieve the terms that match using this path

3. Apply the terms that don't match the index (if any)
- Other terms are used to discard some retrieved tuples

General Selection Conditions:
- First the selection is converted to <conjuctive normal form (CNF)>
e.g.
(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3
is converted to:
(day<8/9/94 OR bid=5 OR sid=3 ) AND (rname='Paul' OR bid=5 OR sid=3)

A B-Tree index matches a conjunction of terms that involve only attributes in a
<prefix of the search key>
- Index on <a, b, c> matches <a=5 AND b=3> but not <b=3>

Hash indexes must have all attributes in search key
// since you can't search for ranges in hash indexes

If we have 2 or more matching indexes:
1. get sets of rids of data records using each matching index
2. Then intersect these sets of rids
3. Retrieve the records and apply any remaining terms.

```
// I think this method is actually really similar to the set reduction used in
// declarative programming project 1? Minus the record retrieving bit.

Example: Consider name="Trump" AND is_president=True AND builds_wall=True
Assuming there is a B+ tree index on name and an index on builds_wall:
1. a) Retrieve rids of records satisfying "Trump" <using the first B+ index>
   b) Retrieve rids of recs satisfying builds_wall=True <using the second index>
2. Intersect
3. Retrieve records and <check for is_president=True>

---------------------- Projection ----------------------

!Projection with Sorting:

[sql]
SELECT DISTINCT R.sid, R.bid FROM Reserves R
[end]

How do we figure out duplicates?

Basic approach is to use <sorting>.

1. Scan R, extract only the needed attributes //just to save memory
2. Sort the resulting set
3. Remove adjacent duplicates

A cost estimation for this would be like this, assuming R has 1000 records:
Step 1: 1000 I/O
Step 2: 250 I/O          // Assuming Reserves has size ratio 0.25 = 250 pages
                         // qsort is O(n log n) but we only calculate I/Os
Step 3: 250 I/O

This cost estimate assumes <everything fits into fast memory>.
But we can't always afford a few thousand terrabytes of RAM :/

So, one way to get around this is with <external merge sort>

!Projection with Sorting Out of Memory (External Merge Sort):
This is a cool thing which allows you to use a sorting algorithm over a very
large amount of data.

Basically it works by sorting the data in parts.

These parts are then iterated parrallel to each other and merged, hence the name

Example of merge sort:

1. Split into smaller parts, then sort then (with like qsort or something)

2. Then look at all the parts parrallel to each other and merge

Where p1, p2 and out are buffers

p1             p2             out
| 1, 2, 5 |    | 1, 2, 3 |     | Empty |
```

```
  ^               ^

p1              p2              out
| 2, 5 |        | 1, 2, 3 |      | 1 |      // in this step, 1 already exists
  ^               ^                          // in output buffer, so discard 1

p1              p2              out
| 2, 5 |        | 2, 3 |         | 1 |
  ^               ^

p1                 p2           out
| 5 |            | 2, 3 |        | 1, 2 |
  ^               ^

p1              p2              out
| 5 |          | 3 |           | 1, 2 |
  ^               ^

p1              p2              out
| 5 |          | ... |         | 1, 2, 3 | // once the buffer p2 runs out, load
  ^               ^                          // more data from the disk

p1              p2              out
| ... |        | ... |         | 1, 2, 3, 5 |
  ^                 ^
```

Note for later:
As you can see, it takes 1 operation to sort the parts, and then another 1 to merge into the out buffer. This makes up the "2" in the cost calculation later.

//I'm not going to explain it because the slides are spaghetti and this guy does
//a way better job:

// Watch them, they're only like 3 minutes each
### https://www.youtube.com/watch?v=wTAVwbvjiac
// I think this one is the one that's in the slides (about 1:30 onwards).

### https://www.youtube.com/watch?v=1kr81S3HIR8
### https://www.youtube.com/watch?v=5mKtLu60pdw

With external sort, the cost becomes this:
Cost with 20 buffer pages:
1. Scan R: 1000 I/O + 250 I/O // scanning/partitioning for external sort
2. Sort: 2 * 2 * 250 I/O
    - The first "2" is for the sorting/merging, which is <done in 2 phases>
      250 pages / 20 buffer -> 13 runs in order to fully sort/merge
      <Remember we only count I/O, so the 2 represents one sort op and one >
      <merge op>
    - The second "2" represents a read/write operation
    - Finally do this to all the pages - 250
3. Remove Duplicates: 250 I/O

Total: 1000 + 250 + 2*2*250 + 250 = 2500 I/Os


!Projection with Hashing:

This is a lot more simple than above, simply hash data into buckets and remove any adjacent duplicates from the buckets.

1. Scan R, extract the needed attributes
2. Hash data into buckets
3. Remove adjacent duplicates from buckets

Cost:
1000 + 250 + 250 = 1500 I/O

But again, we can't always fit it in memory,
so

**!Projection Hashing out of Memory:**
1. Partition data into B partitions with h1 hash function
- Hopefully now a partition will fit into memory.

2. Go through each partition and hash each page with the h2 hash function

3. If a oversized partition is encountered, recursively partition this with step 1.

4. Check for duplicates in each h2 bucket

Cost is weird:
1. Read: 1000 I/O
2. H1 Write: 250 I/O
3. Read H1 Partitions: 250 I/O

Total 1500 I/O

---

Sort-based approach is standard, doesn't have the problem of <skewing> with hash and the <result is sorted> which is a nice bonus.

If there is <enough memory>, both have <same I/O cost>, as calculated by

M + 2T

Where:
M = pages in R
T = pages in R with unneeded attributes removed.

---

## Query Processing (Joins)

---

**!Joins:**

- Really common operation
- Can be super <e x p e n s i v e> like cross products.

Here's some implementation techniques:
- Nested-loops join
- Index-nested loops join

- Sort-merge join
- Hash join

For the following examples, we have two Relations S and R.
S - tuples are 50 bytes long, 80 tuples per page, 500 pages
    N = 500, $p_s$ = 80
R - tuples are 40 bytes long, 100 tuples per page, 1000 pages
    M = 1000, $p_r$ = 100

R will be the outer relation and S will be the inner relation.

---------- Nested Loop Joins ----------

**!Simple Nested Loops Join:**
- literally in the name, <for each item, iterate through the relation and check>

Pseudocode:
```
for i in R:                 // p_r*M = 100*1000      Because we load every tuple
  for j in S:               // N = 500               Here we only load the page
    if i==j:                //                       once into fast memory
      result.append(i,j)    // M = 1000              Output
```

Cost in I/Os:
$(p_r * M) * N + M = 100*1000*500 + 1000$
 ^ note that this is <the outer relation>, so it could be $p_s$ if S was outer.

**!Page-Oriented Nested Loops Join:**
- Basically do simple nested loops join, but on <every single page in both R & S>

Pseudocode:
```
for r_page in R:                // M = 1000
    for s_page in S:            // N = 500
        for i in r_page:        // These don't count because we loaded the pages
            for j in s_page:    // into fast memory
                if i==j:
                    results.append(i,j)     // M = 1000
```

This is <cheaper than doing a simple nested loop join>, as only the corresponding page needs to be loaded instead of the whole relation.

Cost:
M*N + M = 1000*500 + 1000

**!Index Nested Loops Join:**
// Note: Apparently this won't be examined because they messed it up in L14

- This is basically a simple nested loop join, HOWEVER it <utilizes an index> if the relation has one.

Pseudocode:
```
for i in R:                 // p_r*M = 100*1000
    j = index_search(i, S) // This can be 1.2 for hash index, or 2-4 for B+ tree
    result.append(i,j)      // M = 1000
```

Cost:
M + ($p_r$*M)*(whichever index you're using + 1) = 1000 + 100*1000*2.2

#In this case, the 2.2 comes from

1.2 I/O to find the record i in S, and 1 I/O to get the matching j record.
// using a hash index


!Block Nested Loops Join:
- This is really similar to page-oriented join, however multiple pages of R are held, which is scanned against S and then output in sections called "blocks"

---------- Sort-Merge and Hash Joins ----------

!Sort-Merge Join:
- Sort R and S.
- Then "merge" them, similar to <the method as described on line 890>, except <only checking for equality instead of discarding equal values>.

Pseudocode:
sort(R)
sort(S)
merge R and S where r==s

Cost:
Sort R + Sort S + (M + N)

- <Less> sensitive to <data skew>
- <Result is sorted>
- <Goes faster if> one or both inputs <already sorted>

!Hash Join:
- Partition the relation the same way as <described on line 950>, except instead <only checking for equality instead of discarding equal values>.

Cost:
Partitioning = 2(M+N)
Matching = M+N

Total = 2(M+N) + M+N

- <Better> if relation <sizes differ greatly>
- Highly <parallelizable>

---------- General Joins ----------

So far we've done NL for equality conditions.

Inequality conditions <must use a B+ tree index>
# Hash Join, Sort Merge Join cannot be used for inequality!!!!

<Block Nested Loop is best for inequality conditions>

!Special Joins:
- Intersection
- Cross products
- Union (Distinct)

For Union:
Sort-merge again
- Sort both relations, scan sorted relations and merge

OR

Hash based
- Repeat hash based process as described on line 950, but discard duplicates and add tuples to output.

**!Aggregate Operations:**
- An aggregate option is an action that results in a single value e.g. AVG, SUM, MIN etc.

Without Grouping (like by a category):
- Aggregate functions <require scanning the relation>
- Can be done with an <index only scan>

With Grouping:
- <Sort on group by attributes>, then <scan relation> and <compute aggregate> for each group.

---

## Query Optimization

---

<Many methods of executing> a query, all giving the same answer.

**!Query Plan:**
This is a <tree> with <relational algebra operators for the nodes>, showing possible <choices of algorithms>.

By convention, <outer is on the left>.

# How do I get gud and optimize f a s t?:
1. Break query into "blocks"
    - Query Block = Unit of Optimization
    - Nested Blocks are treated as calls to a subroutine, made once per outer tuple

2. Convert each block <into relational algebra>

3. Consider all the <query plans> for <each block>
    - <Every query> has a <core SPJ select-project-join> expression
    - Group By, Having, Aggregation and Order By are <all done after SPJ>
    - <Focus on optimizing SPJ core>

4. Plan with the <lowest estimated cost is selected>

**!Equivalence:**

Selection <can be done condition by condition>: that is, if I ask for a name and age, you can <return age and then name in any order>

Projection, however, <must be done with the specified conditions all at once>

**!De-Correlation:**
- Convert correlated subquery into uncorrelated subquery

Basically translating subqueries so that if they are in a nested for loop, they only need to be executed once:

[sql]
```
SELECT S.sid
FROM Sailors S
WHERE EXISTS
  (SELECT *
  FROM Reserves R
  WHERE R.bid=103
  AND R.sid=S.sid) /* This is redundant */

/* vs */

SELECT S.sid
FROM Sailors S
WHERE S.sid IN
  (SELECT R.sid
  FROM Reserves R
  WHERE R.bid=103)
```
[end]

**!Flattening:**
Basically using a join algorithm + optimizer instead of two select-projects
e.g.

[sql]
```
SELECT S.sid
FROM Sailors S
WHERE S.sid IN
    (SELECT R.sid
    FROM Reserves R
    WHERE R.bid=103)

/* vs */

SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
AND R.bid=103
```
[end]

**!Cost Estimation:**
For each plan, estimate cost:
- Estimate the cost of each operation in plan tree
- Estimate the size of result for each operation

Performance Estimation:
# I/O + factor * CPU instructions

Size Estimation:

```
# Result = Max Number of Tuples * product of all RF's

RF = Reduction Factor, where

for equality, RF = 1/Keys(I)
for range,    RF = (High(I)-value)/(High(I)-Low(I))

Assume RF = 1/10 otherwise

Estimated Size for Joins:
Depending on which relation has more tuples,

est_size = NTuples(R) * NTuples(S) / MAX(NKeys(S), NKeys(R))

// Remember,

// What is query optimization/describe steps?
// Equivalence classes
// Result size/cost estimation

// is examinable and is probable a big part of it.

// try not to die

!Enumeration of Alternate Plans:

Single-relation plans

Multiple-relation plans

Single Relation Plans:

-------------------------------------------------------------------------------
                                Normalisation
-------------------------------------------------------------------------------

All about the process of <abstraction> and <removing undesired redundancy>.

Not normalising tables will cause <anomalies>.

<Anomalies are not good!>

!Anomalies:
# Insertion Anomaly
    e.g. A new course <cannot be added> until at least <one student has enrolled>
         - Which comes first? Student or course?
# Deletion Anomaly
    e.g. If a <course only has one student>, and that <student withdraws>, we
        <lose all information regarding the course he took>!
# Update Anomaly
    e.g. If the <student appears multiple times> and he changes year level, then
        we have to change <all the records containing that student>, otherwise
        we will <have an update anomaly>, where the data regarding an entity is
        <not certain>.

Basically just look for <repeating data> or <data that represents a separate entity>
```

!Functional Dependancy:
- A <functional dependency> concerns <values of attributes in a relation>.
- A set of attributes <X determines> another set of attributes <Y> if each value of <X is associated with only one value of Y>.

Written as: X -> Y  // Similar to y = f(x)
EmployeeID -> Name
EmployeeID -> Salary

X is a <determinant> (an attribute on the left hand side of the arrow)

As stated before, all attributes are <either part of the primary key> <or not>.

A functional dependency can be <partial> if <one or more non-key attributes> have <a dependency on part of the primary key>.
e.g.
If the dependency is X,Y -> Z
then the partial dependence is Y -> Z or X -> Z

A functional dependency is <transitive> if <it is a dependency between 2 or more> <non-key attributes>.

---------- How to Normalise Real Good ----------

!First Normal Form (1NF): Remove Repeating Groups

<Repeating groups of attributes shouldn't> be represented <in a flat table.>
Removing cells with multiple values will solve this!
e.g.
// Let's start with something flat. I'll use <> for underline
Order-Item(<OrderID>, CustomerID, <ItemID>, Desc, Qty)
// Notice how item has it's own attributes?
Order-Item(<OrderID>, CustomerID, (<ItemID>, Desc, Qty))
// Split into two
Order-Item(<OrderID, ItemID>, Desc, Qty)
Order(<OrderID>, CustomerID)
// Yay we're kinda normal now

!Second Normal Form (2NF): Remove partial dependencies

A <non-key attribute> <cannot be identified> by <part of a composite key>.

e.g. //cont.
Order-Item(<OrderID, ItemID>, Desc, Qty)
// Notice how Desc is identified by ItemID, which is part of the composite key.
Order-Item(<OrderID, ItemID>, Qty)
Item(<ItemID>, Desc)

Note that after we've done this, <we've solved most of the anomalies>.
1. Updating an <Item> will change the Desc wherever the Item is referenced
2. Data for an <Item> is <not lost when the last order for that item is deleted>
3. You can <add items without having to make an order for them>

!Third Normal Form (3NF):  Remove transitive dependencies

A non-key attribute <cannot be identified> by another non-key attribute.
e.g.
Employee(<EmpID>, Ename, DeptID, Dname) // Look for foreign keys
// In this case, Dname is identified by DeptID, so we normalise it
Employee(<EmpID>, Ename, <DeptID>)
Department(<DeptID>, Dname)

// this isn't even my final form
!Boyce-Codd Normal Form:  Ensure determinates are candidate keys
"<Every non-key attribute> must <provide a fact about the key>, the whole key,
and nothing but the key. (So help me Codd)"
// woah this lecturer is way more fun

e.g.
Allocation(<StudentID, Subject>, Teacher, GPA)
// GPA is only linked to StudentID, however if Teacher is made part of the key,
// then GPA provides information about the teacher and the student, since
// it assume a student can take a subject more than once
Allocation(<StudentID, Teacher>, Subject)
Assignment(<StudentID, Teacher, GPA>)
Department(<Teacher>, Subject)

---

However, <there is a pay-off> to having all the relations being neat and tidy with
    no anomalies:

<Denormalized tables> are <f a s t e r>, but <more work must be done> to keep the
    database consistent.
Denormalization <can be used to improve performance of time-critical things>

Normalised relations however <contain a minimum amount of redundancy> and allows
    users to <insert, modify and delete> freely <without errors>.

---

## Database Administration

---

The "Database Administrator" role  // Only unlocked in New Game+

4 responsibilities as part of the <DBA role>
Capacity Planning:
    - Estimating <disk space> and <transaction load>
Performance Improvement:
    - Common approaches
    // phat gains
Security:
    - Threats
    - Web apps and SQL Injection // look this up 2 be a cool h4x0r
    // also firearms training and CQC

Backup and Recovery:
    - Types of failures, responses to these, types of backups
    - Other measures to <protect data>

Primarily concerned with "maintenance" / "ops" phases
- But really <should be consulted> during <all phases of development>

DBA can be made <redundant by cloud-based DBMS>

**!Capacity Planning:**
Process of <predicting when future load levels will saturate the system> and
    determining the <most cost-effective way> of <delaying system saturation> as
    much as possible.

Need to consider:
- Disk space requirements
- Transaction throughput
// how much alcohol you'll need when the servers go down

Estimating Disk Space Requirements:
- Treat database size as the sum of all table sizes
- where table size = number of row * average row width

Need to know the size of values. This is dependent on the storage engine used.
Go look at the documentation!
To calculate the <width of the row>, simply <add up all the datatype sizes>

**!Estimate growth of tables:**
Pretty simple, just gather estimates during <system analysis>.
e.g. "The company sells 1000 products. There are 2 million customers who place
        on average 5 orders per month. An average order is for 8 different
        products".

will make:
Product ->    1000 rows
Customer ->   2,000,000 rows
Orders ->     10,000,000 rows
OrderItems -> 80,000,000 rows

Figure out storage per year according to the new data.
Note that "Event" tables grows a lot faster than "entity" tables, so we can put \
    event tables on separate disks.

**!Estimate transaction load:**
- Consider each business transaction.
- How often are these run?
- For each transaction, what SQL statements will be run?

Multiply all of these and then you get the transaction load.

Things to consider:
- Differentiate <peak vs average load>.
    e.g. a sale will result in <many more transactions>
- Acceptable <response time>
    - We can <reduce the response time> by:
        - Application/Database <Design>

```
        - <Tuning> DBMS/individual queries
        - <not running everything on a potato>
- Availability
    - 9-5 vs 24/7
    - Who's the customer? e-business? Do they need this all the time?

What affects database performance?:
- Caching data in memory
- Placement of data files
- Database replication and server Clustering
- Use of fast storages such as SSDs
- Use of indexes to speed up searches/Joins
    - You must decide <when to create indexes>
        - Frequently queried columns
        - Primary/Foreign Keys / Unique Columns
        - Large tables
        - Columns with a wide range of values
- Choice of data types
- Program logic
- Query execution plans
- Good code (no deadlocks)

!Security:

Threats to Databases:
Loss of Integrity:
- Keep data consistent
- Free of errors/anomalies

Loss of Availability:
- Must be available to authorized users for authorized purposes

Loss of confidentiality:
- Must be protected against unauthorized access

We can <protect databases> by:
- Access Control
- Encryption

!Access Control:
All <about restricting access> to data and <granting and revoking> privileges
This <is required by the security mechanism of a DBMS>

- Access control is handled by the admin creating user accounts
- Database keeps a usage log
- When <tampering is suspected, perofrm an audit>.
- Need to control <online and physical access> to the database

Types of Privileges:
- Account Level
    DBA specifies the particular privileges that each user holds
    i.e. this user <can do x> on the database
- Table Level
    Controls a user's privileges to <access particular tables> or views
- Schema Level
    Controls a user's privileges to <access a particular scheme> in the database
```

Views are pretty important since you can <hide database structure and data>

You can also <encrypt> things to <protect sensitive data> whenever they are
    transmitted over a network
- This <prevents interception by third party>

Encrypt data in the database
- Provides some protection in case of unauthorized access.

**!SQL Injection:**
// this is fun
A technique used to exploit web applications that use user unput within
    database queries.

Basically you can <run queries in fields like username and password>.

### https://xkcd.com/327/

One example of a h4x0r injection

usr: ' or 1=1; --
pw: any text

will result in:
[sql]
```
SELECT * FROM User
WHERE username = '' or 1=1; -- ' and password = 'any text';
```
[end]
The ' closes the quotes, then there is an always true condition, and the rest
of the query is commented out.

**!Backup and Recovery:**
This is pretty self explanatory

backup ur shit boi

Types of Backup:
# Physical Backup
- Raw copies of Files and Directories
- Suitable for large databases that need fast recovery
- Database is preferably offline ("cold" backup) when backup occurs
- Backup = exact copies of files
- Backup <should include logs>

# Logical Backup
- Backup <completed through SQL Queries>
- Slower than physical
- <Output is larger than physical>
- <Doesn't include log or config files>
- Machine independent
- Server is available during the backup

You can backup Online - "HOT" or Offline - "COLD"

Online Backups are done <while the database is live>.

Offline backups are done <while the database is offline>.
- Offline is preferable
- Simpler to perform
- Backups occur when the database is stopped

You can backup Fullly or Incrementally.
Fully contains the <whole database>                    // livin in the databse
Incremental <only contains changes> since the last backup

You should have a <backup strategy> which is a <combination of full/incremental>
    backups

You should also have <offsite backups> just in case your building explodes or
    something

Should also use <Server Replication>, <Server Cluster> and <RAID> // love RAID
// RAID 0 FOR FULL BACKUP POTENTIAL (not really)

Replication:
- One writer/Many readers
- Some protection against server failure
- Multiple copies of data
- Replicates accidental data delection

Clusters:
- Automatic synchonous partition
- Protection against server failure
- Multiple copies of data

RAID:
- Software or Hardware RAID
- Depends on the RAID level // raid 0 will be f a s t but won't help backup

!Categories of Failure:

Statement Failure:
- Bad syntax

User Process Failure:
- The process doing the work fails

Network Failure:
- Network failure between the user and the database

User Error:
- User accidentally drops the rows, table, database
// me_irl

Memory Failure:
- Memory fails, becomes corrupt

Media Failure:
- Disk failure, corruption deletion
// i.e. your computer catches fire

---

## Transactions

---

**!Transaction:**

<The system> should <ensure that> updates of a <partially executed transaction>
    are <not reflected in the database>.

A good transaction requires #ACID

**!ACID:**
Stands for:
- Atomicity
    - A transaction is a <single, indivisible, logical unit of work>.
    - <All operations> in a transaction <must be completed>; if not, then the
        transaction is aborted.

- Consistency
    - <Data constraints> that hold <before> a transaction must also hold <after it>
        e.g. PKs and FKs and implicit constraints (sums etc.)

- Isolation
    - <Data used during execution> of a transaction <cannot> be used by a second
        transaction <until the first one is completed>
    - Solved by <running transactions serially> (one after another).

- Durability
    - When the <transaction is complete> the changes made are <permanent> even
        if the <system fails>

If a transaction has <multiple statements>, they are usually <preceded> by a
<START TRANSACTION> or <BEGIN> and <ended> by a <COMMIT>

```sql
START TRANSACTION; -- (or, 'BEGIN')
SQL statement;
SQL statement;
SQL statement;
…
COMMIT;
```

ROLLBACK is used to <undo everything> (in case the transaction doesn't finish)
Transactions are "all or none"

**!Concurrent Access:**
When <multiple users> execute DML (Database Modification Language) against a
<shared database>

Can result in <uncommitted data> if two queries are <executed at the same time>

e.g. If you change the bank account of someone <while they're withdrawing>, then
        <commit the changes before the other person finishes>, then you <do not>
        record changes of <both executions>.

It'd be like <item duping> if one person held an item while another person drops it from the inventory

**!Serialisation:**
We can <serialise> a transaction by ensuring all executions are done <one after> another

However this is <real slow> and <not realistic>

So we have <concurrency control>

**!Concurrency Control:**
- Locking/Version Control

**!Locking:**
- <Guarentees exclusive use> of a data item to a <current transaction>
- <Required to prevent> another transaction from <reading inconsistent data>

The <Lock Manager> is required for assigning/policing the locks

Database Level Lock:
- Entire database is locked
- Good for batch processing but unsuitable for multi-user DBMSs

Table Level Lock:
- Entire table is locked
- Multiple users can use the database as long as they <use separate tables>
- Can cause bottlenecks
- Not suitable for highly multi-user DBMSs

Page Level Lock:
- Page is locked

Row Level Lock:
- <Most Popular>
- Locks only the row of a table
- Improves <data availability> but with <high overhead>

Field Level Lock:
- Most flexible lock but requires <extremely high level of overhead>

Locking can result in a <dead lock>, where two transactions wait for each other to unlock data.

Prevent this by locking potential records in advance - detect/end the transaction

**!Versioning:**
- <Timestamping> all the transaction
- Each <transaction attempts an update>
- System will review and <reject inconsistent updates>

---

## Data Warehousing

---

**!Data Warehouse:**

- A <single repository> of <organisational data>
- Integrates data from mulitple sources

- Pretty much a <informational database>

Transactional vs Informational:
- a Transactional system's purpose is to <run the day to day business>, whereas the informational system's purpose is to <support decision making>

A transactional database represents the <current state of business> whereas the informational database represents <historical data>.

This is also reflected by the users - customers/employees will use transactional while <managers and analysts> use <informational>

**!Dimension Modelling (Star Schema Design):**
Consists of:
- A fact table
- several dimensional tables
- Hierarchies in the dimensions

Essentially a <simple and restricted> type of ER model.

**!Fact Table:**
- A fact table contains the actual business measures called facts
- Also contain foreign keys for dimensions
- Really really simple ER model

---

## NoSQL

---

As much as the relational database we've learnt about is <flexible> and <fast>, relational dbs are <not good with big data> and <clustered/replicated servers>

**!NoSQL Database:**
- Doesn't use relational model
- Runs well on <distributed serves>
- <Scaling out> rather than <scaling up>
- <Open Source> usually
- Built for <modern web> and <natural for cloud environment>
- <Schema-less>
- Not ACID compliant

Built to <handle larger data volumes>

Some examples of <NoSQL> are <JSON> and <XML>.

Because NoSQL languages are <built around big data>, they just capture everything in a <data lake> and <worry about structure later>.

**!Data Lake:**
- A large integrated repository for internal and external data that <doesn't follow a predefined schema>
- Capture <e v e r y t h i n g>

Usually <structuring and organizing the data> takes place <during data analysis>

There are many different types of NoSQL:
# Key-Value store
    e.g. JSON
# Document Databases
    Examinable/Queriable key-value dbs
    e.g. MongoDB, CouchDB, Terrastore, OrientDB, RavenDB
# Column Families
    Columns are stored <instead of rows>
    Kind of like automatic vertical partitioning
    e.g. DynamoDB, Cassandra
# Graph Databases
    Node-and-arc network
    Difficult to program in relational DB
    Stores entities and their relationships
    e.g. Neo4j

**!CAP Theorem:**

PICK TWO:
# Consistency:
    - Everyone always sees the same data
# Partition Tolerance:
    - System stays up when network between nodes fail
# Availability:
    - System stays up when nodes fail

Or alternatively, IF a partition occurs, <you must choose Consistency or Availability>

// godspeed guys
// im pretty dead
// o7