

# End o' semester clean up



A little bit of everything

# Database Optimization

Two approaches... what do you think they are?

# Improve the Hardware

- Has been a great solution in recent decades, thanks Moore!
- Throwing more hardware at a problem (typically) helps... but makes lazy programmers
- Databases can make use of all sorts of hardware upgrades
  - Multi-core CPUs
  - RAID hard drives
  - RAM

# Improve the Software

- **SQL tuning** → writing better queries
- **DBMS tuning** → making the system respond to the same query faster

# The DBMS Side

- There are many architectural choices to make:
  - How large to make your files for storage? Where and how do you store them? Which file system is most appropriate?
  - How much RAM and cache can you dedicate towards the DBMS?
  - How can I/O be minimized?
- Often these decisions are tied to the hardware available

# The SQL Side

- Every query goes through these stages:
  1. Parsing → how the query gets interpreted
  2. Execution → Run! Figure out the answer.
  3. Fetching → Get the actual data and return it

# Parsing

- Each query is interpreted, much like other programming code
  - Validate correctness (syntax, names, etc.)
  - Validate permissions
  - Decomposition of the query into atomic parts
  - Optimization (if any)
  - Define an execution plan, or how the result set will be figured out by the DBMS. These can be scan the table, grab a row, grab a cell, merges, sorts, etc.

# Indexes

- These make an immense difference on performance because they direct the DBMS where exactly to get the needed data
- **Sparsity** → how many different values a column can have. Note that this is *not* the number of rows, but the number of possible values.
- Depending on the sparsity of a column an index may or may not speed up queries
- MS SQL can help you with this!

# Programmer Choices

- For indexes these are good things to look for:
  - If a column is by itself in a WHERE, HAVING, GROUP BY, or ORDER BY
  - When MAX or MIN is used on the column
  - When data sparsity is high or large tables
  - When columns are used in joins that aren't already marked PK or FK (aside: why not PK and FK?)
- Properly declaring PKs and FKs similarly helps the interpreter

# Clever Conditionals

- Much of the rest of the advice applies not only to SQL but programming in general
- All of these are 'if you can' situations
- Avoid calculations, function calls, and regular expressions in comparisons
  - Ex `price < 10` is fast but `qoh > min*10` is slow
  - `Name = 'Alex'` better than `Name = UPPER( 'Alex' )` which is better than `Name = 'Alex%'`
- Numeric comparisons are faster than any other, with integers being faster than decimals

# Clever Conditionals

- IS NULL and IS NOT NULL comparisons are very slow because nulls are not stored in indexes
- Equality is faster than inequality
  - In a AND/OR string of comparisons, place equalities before inequalities
- Eliminate calculations
  - $\text{Price}/2 > 10$  should be coded as  $\text{Price} > 20$

# Clever Conditionals

- With multiple AND conditions, order them by likelihood of being false
- With multiple OR conditions, order them by likelihood of being true
- Eliminate NOT by changing conditionals or using De Morgan's Laws
  - NOT( sex = 'M' ) is the same as sex = 'F'
  - NOT( price > 10 ) is the same as price <= 10
  - NOT( sex = 'M' AND price > 10 ) is the same as NOT( sex = 'M' ) OR NOT( price > 10 ) which becomes sex = 'F' OR price <= 10

# Newer ideas in databases

- You've spent all these time learning about normalization and SQL and now....

# No SQL!

- Relational databases are not a new idea, maybe there's more up-to-date theory
- The latest and greatest is NoSQL or schema-less databases
- They give up joins and ACID but gain ease-of-use and speed

# The idea

- With traditional relational databases, data is broken down so it may be modeled into tables, an appropriate level of normalization is chosen, and data is gathered through joins
- ERDs are typically created and table structure is specified (thus creating the schema) at time of creation, which is a pain to change later

# The Idea

- Schemaless says do away with the schema and data separation
  - Similar data is stored together in documents
  - It is very similar vein to that of OOP
  - Because of data organization, simple joins are unnecessary

# Why is it good?

- Related data, because of being in the same document, is stored together
- Much easier to keep more data in RAM
- No joins mean the DB is more optimized for data retrieval
- Faster in general
- No SQL injection worries
- Simple replication/sharding
- No need for object-relational mapping, a.k.a. “the Vietnam of Computer Science”

# Why is it good?

- Schemaless!
  - Easier conceptually
  - Add or remove data as needed without locking
  - Null values are no problem (well, almost...)
- Efficient storage of files
- Scales well
- Generally considered more productive
- Data/object embedding as well as links

# Why is it bad

- Giving up ACID...
  - No transactions is the biggest worry
  - No guaranteed consistency, usually just *eventual* consistency
  - Isolation... eh
  - Durability... you never really get it anyway
- No SQL
  - Sometimes people just want to use joins
  - May have a large codebase anchored to SQL
  - Asking for data is mostly the job of the application programmer... depending on the language this may be easy or hard

# SQL and the Web

- There's not a whole lot of fanciness going on here
- Most common setup is LAMP: Linux-Apache-MySQL-PHP

# Getting some data

- What's HTML exactly?
- `<form action="insert.php" method="post">`  
First Name: `<input type="text" name="first"><br>`  
Last Name: `<input type="text" name="last"><br>`  
Phone: `<input type="text" name="phone"><br>`  
`<input type="Submit">`  
`</form>`

# Use PHP to process it

- ```
<php?
$username="username";$password="password";
$database="your_database";$first=$_POST['first'];
$last=$_POST['last'];
$phone=$_POST['phone'];
$email=$_POST['email'];
mysql_connect(localhost,$username,$password);
@mysql_select_db($database) or die( "Unable to
select database");
$query = "INSERT INTO contacts VALUES
('','$first','$last','$phone','$mobile','$fax','$email','$web')
";
mysql_query($query);
mysql_close();
?>
```

# Use PHP to display a query result

- <?>

...connection stuff the same....

```
$query="SELECT * FROM contacts";
```

```
$result=mysql_query($query);
```

```
$num=mysql_numrows($result);
```

```
mysql_close();
```

```
echo "<b><center>DB Output</center></b><br><br>";
```

```
$i=0;
```

```
while ($i < $num) {
```

```
  $first=mysql_result($result,$i,"first");
```

```
  $last=mysql_result($result,$i,"last");
```

```
  $phone=mysql_result($result,$i,"phone");
```

```
  echo "<b>$first $last</b><br>Phone:"
```

```
  $phone<br><hr><br>";
```

```
  $i++; } ?>
```

# SQL Injection

- Remember how the NoSQL guys aren't vulnerable to this?
- What is it?
- Why is it a big deal?