Engineering Robust Server Software

Scalability
Other Scalability Issues

• Database
• Load Testing
Databases

• Most server applications use databases
  • Very complex pieces of software
  • Designed for scalability

• …but how well depends on what you are doing with them…
First Question: DB or in Your Program?

- Do you do computation in DB or in your own code?
  - Tradeoffs
    - Code to write/test
    - Costs of consistency
    - IO costs / stored data
  - Might be good to mix the two
    - E.g., hwk4 you might store transactions in db, but match in memory
Databases and Concurrency

• How do databases handle concurrency?
  • Could use locks, but… what we learned about those?
    • (they’re expensive)
  • Postgres (and many others): Multi-version concurrency control (MVCC)
    • Basically, the DB keeps multiple versions
    • Ensures consistency based on transaction isolation level
Serializability

• Serializability: the results of a concurrent execution of operations are equivalent to executing the operations in some sequential order
  • e.g. A + B + C concurrently produces results equivalent to executing…
    • A, B, C or A, C, B or B, A, C or B, C, A or C, A, B or C, B, A
  • Benefits?
    • Enables parallel operation but matches programmer’s intuition
• Does this sound similar to any other ideas we have learned recently?
  Atomics and memory consistency!
• What do you think we might do?
  Use different consistency models.
First, databases have transactions

- Transactions are like critical sections, but for databases:
  All statements in a transaction either all happen or none of them

```
BEGIN;
  SELECT count FROM tbl WHERE id = 42;
  INSERT INTO tbl VALUES (8, 12);
  SELECT count FROM tbl WHERE id = 67;
  COMMIT;
```
Transaction isolation

- Transaction isolation: How much are the effects of two transactions allowed to interleave?

- Strictest model: **Serializable** ("it’s like they ran atomically and sequentially")

BEGIN;
SELECT * FROM tbl WHERE id = 42;
INSERT INTO tbl VALUES (4, 7);
SELECT * FROM tbl WHERE count<10;
COMMIT;

BEGIN;
SELECT * FROM tbl WHERE id = 42;
INSERT INTO tbl VALUES (5, 9);
SELECT * FROM tbl WHERE count<10;
COMMIT;

Concurrent change visible? **No!**
Isolation Levels

There are more isolation levels than serializable, each less strict than the last. Sorted by which kind of “weird things” are allowed to happen.

• **Serializable isolation**
  • Nothing unexpected

• **Repeatable Read isolation**
  • Can have "phantom reads"

• **Read Committed isolation** (default in Postgres)
  • Can have "non-repeatable reads" (+phantoms)

• **Read Uncommitted isolation**
  • Can have "dirty reads" (+non-repeatable +phantoms)
Weird thing: Non-Repeatable Read

```
<table>
<thead>
<tr>
<th>id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>57</td>
</tr>
<tr>
<td>67</td>
<td>128</td>
</tr>
<tr>
<td>99</td>
<td>0</td>
</tr>
<tr>
<td>456</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Non-Repeatable Read: Values within a row change between reads

```
SELECT count from tbl WHERE id = 42;
82

UPDATE tbl SET count = 57 WHERE id = 42;
COMMIT;

SELECT count from tbl WHERE id = 42;
57
```
Weird thing: **Phantom Read**

<table>
<thead>
<tr>
<th>id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>57</td>
</tr>
<tr>
<td>67</td>
<td>128</td>
</tr>
<tr>
<td>99</td>
<td>32</td>
</tr>
<tr>
<td>456</td>
<td>1</td>
</tr>
</tbody>
</table>

**Phantom Read:** Set of rows in a query change between reads

```sql
SELECT * from tbl WHERE count < 10;
(99,0)
(456,1)

UPDATE tbl SET count = 32 WHERE id = 99;
COMMIT;

SELECT * from tbl WHERE count < 10;
(456,1)```
Weird thing: **Dirty Read**

<table>
<thead>
<tr>
<th>id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>57</td>
</tr>
<tr>
<td>67</td>
<td>128</td>
</tr>
<tr>
<td>99</td>
<td>32</td>
</tr>
<tr>
<td>456</td>
<td>77</td>
</tr>
</tbody>
</table>

**Dirty Read:**
Read data from an *uncommitted* transaction

```sql
UPDATE tbl SET count = 77 WHERE id = 456;
ROLLBACK;
```

```sql
SELECT count from tbl WHERE id = 456;
77
```
Isolation Levels in Postgres

There are more isolation levels than serializable, each less strict than the last. Sorted by which kind of "weird things" are allowed to happen.

- **Serializable isolation**
  - Nothing unexpected
  - Can throw exns for violations

- **Repeatable Read isolation**
  - Can have "phantom reads"
  - Can throw exns for violations

- **Read Committed isolation** (default in Postgres)
  - Can have "non-repeatable reads" (+phantoms)

- **Read Uncommitted isolation**
  - Can have "dirty reads" (+non-repeatable +phantoms)
  - Not actually available: upgraded to Read Committed

For more on isolation in Postgres, see [here](#) (good overview!)
Query Performance

• Many things can affect query performance
  • Complicated topic..
• But how can you gain insight into what is going on?
• Can you do anything to improve it?
  • First thing we always want to do?
Understand Behavior: Explain

explain select * from grades where grade < 62;

QUERY PLAN

-----------------------------------
Seq Scan on grades  (cost=0.00..494.80 rows=55 width=35)
Filter: (grade < 62)

Startup Cost
Total cost (arbitrary units)

• Want to know how your query is going to be executed?
  • Ask Postgres to EXPLAIN it
  • https://www.postgresql.org/docs/current/sql-explain.html
Seq Scan?

- Sequential Scan = linearly examine each element.
  - Sound good?
- No! We can do better..
- How? Ask postgres to build an index

```
CREATE index ON grades (grade);
```

explain select * from grades where grade <62;

```
QUERY PLAN
---------------------------------------------------------------
Bitmap Heap Scan on grades  (cost=4.71..128.27 rows=55 width=35)
  Recheck Cond: (grade < 62)
  ->  Bitmap Index Scan on grades_grade_idx  (cost=0.00..4.70 rows=55 width=0)
    Index Cond: (grade < 62)
```
Indexes

• Why not index everything?
  • Cost to maintain index
  • Building=expensive: do before deploying

• Build indexes that are useful for the queries you need

• See https://www.postgresql.org/docs/current/sql-createindex.html
NoSQL

• Wide variety of databases that don't do SQL
  • Key-value stores
  • Graph-based
  • ...
• Many make tradeoffs to increase scalability
  • Eventual Consistency—may not get most current data
    • CAP theorem (only 2 of Consistency, Availability, Partition Tolerance)
  • Accept possibility of data loss
  • ...
Load Testing

• All of this discussion of scalability..
  • How do we know how well we are doing?
  • Well, then again, how do you know how you are doing for anything?

• Test your code!
  • What is the purpose of testing? Discover problems
  • What is a successful test case? One that shows a problem

• So for load testing, what is our criteria for success?
  • Identify performance/scalability problems
Load Testing

- Rule 1: generate a lot of load
  - Sending one request, then another serially? Not enough
  - Need multiple programs/threads/systems generating load

- Rule 2: system needs significant data to start
  - Why?
Data Size Matters

• Suppose you have 10 rows in a table
  • Does indexing matter? No (probably minorly counter productive)
  • What level of the memory hierarchy do you hit? L1 cache

• Suppose you have 10,000,000,000 (10B) rows in a table
  • Does indexing matter? Yes
  • What level of the memory hierarchy do you hit? Disk

• How different are these performance characteristics?
  • Bandwidth?
  • Latency?
Load Testing

- Rule 1: generate a lot of load
  - Sending one request, then another serially? Not enough
  - Need multiple programs/threads/systems generating load
- Rule 2: system needs significant data to start
  - Performance characteristics depend on size
- Rule 3: data needs to have reasonable characteristics
  - Match values/conditions on values of real data
  - Why?
Data Must Be Realistic

- Suppose you run the query
  - SELECT * from whatever WHERE x < 100 AND x > 50;
  - You have only 5 in that range in your test data
  - Your real data ends up with 5,000,000 in that range
- How similar will your performance characteristics be?
Load Testing

- **Rule 1**: generate a lot of load
  - Sending one request, then another serially? Not enough
  - Need multiple programs/threads/systems generating load
- **Rule 2**: system needs significant data to start
  - Performance characteristics depend on size
- **Rule 3**: data needs to have reasonable characteristics
  - Match values/conditions on values of real data
- **Rule 4**: mix and match many combinations of operations in parallel
  - Why?
Mix and Match Operation

• Obvious: we want to ensure each operation done at least once
  • Just like statement coverage.
• But why mix and match them?
  • Different resource usage: cache, bandwidth, ....
  • Different pairings = different resource contention
    • And different DB contention
    • Read by itself vs waiting for a write to commit
What Is "Passing"?

- Ok, so you follow all my rules...
- Make your test cases...
- Run them....
- How do you know if you "passed" the test?
  - ...It depends.... (oh man, I love that answer).
Different Goals

• We might have different goals:
  • Can our system handle the demand from X users?
    • e.g., can DukeHub handle registration?
  • Did we just make it better?
    • e.g., we think we optimized the code, did it really improve?
  • Does our system scale sufficiently with more hardware?
    • Note: requires definition of "sufficiently"
  • Does our system degrade gracefully with more load?
    • Note: requires definition of "gracefully"
Can We Handle Demand of X Users

- Load test with loads that try to mimic X Users
  - May not be hitting system as hard as you possibly can
  - Probably want to add some margin for error

- Measure latencies of requests
  - See how many are within tolerable range
    - Define tolerable?
  - Quite possibly in terms of % guarantees
    - e.g., 99% of requests took less than 500 usec.

![Graph showing latency distribution](image)
Did We Make It Better?

- You do something to your code to improve scalability
  - (Add an index, replace a locked data structure with a lockfree one, …)
  - How do you know it is actually better?
  - Run the old, run the new, measure performance -> see which wins
    - Is it that simple?

*Side note: related to how you convince your boss that*

(a) it was worth your time

(b) he/she should give you a raise for your hardcore hacking
Did We Make It Better?

![Bar chart comparing time (lower is better) for Test1, Test2, Test3, Test4, and Test5, with 'Old' and 'New' categories. The chart shows that 'Old' times are generally higher than 'New' times for all tests.]
Did We Make It Better?

- You do something to your code to improve scalability
  - (Add an index, replace a locked data structure with a lockfree one, …)
  - How do you know it is actually better?
  - Run the old, run the new, measure performance -> see which wins

- Yes, but:
  - Different tests may show different results
  - Different metrics may show different results
    - E.g., slower with this hardware, but more scalable with more hardware
Is Our System Scalable "Enough"?

• What is scalable enough?
  • That also depends…

• How much hardware do we need to add for X more users?
  • Combines two notions of scalability we saw earlier
  • Why does this make business sense?
    • Compute costs money, users bring money -> profitable?
    • Think Cloud Computing
Wrap Up

• Today
  • Databases
  • Load Testing

• Next Week:
  • IO Performance