Transactions
Stats (Out of 90+2)
- Median: 76.5, Mean: 73.9, High: 92, Standard Deviation: 10.5
- Regrade requests open for 1 week (next Tuesday)

Feedback Themes
- TAs were ultra-responsive during the “24 hours”
- Test was too long. We’ll recalibrate for Finals
- Most liked 24 hour flex time
**SQL CHEAT SHEET** http://www.sqltutorial.org

**MANAGING TABLES**

**CREATE TABLE** `t` (
   `id` INT PRIMARY KEY,
   `name` VARCHAR NOT NULL,
   `price` INT DEFAULT 0
); Create a new table with three columns

**DROP TABLE** `t`;
Delete the table from the database

**ALTER TABLE** `t` ADD column;
Add a new column to the table

**ALTER TABLE** `t` DROP COLUMN `c`;
Drop column `c` from the table

**ALTER TABLE** `t` ADD `constraint`; Add a constraint

**ALTER TABLE** `t` DROP `constraint`;
Drop a constraint

**ALTER TABLE** `t1` RENAME TO `t2`;
Rename a table from `t1` to `t2`

**ALTER TABLE** `t1` RENAME `c1` TO `c2`;
Rename column `c1` to `c2`

**TRUNCATE TABLE** `t`;
Remove all data in a table

**MODIFYING DATA**

**INSERT INTO** `t`(`column_list`)
VALUES(`value_list`);
Insert one row into a table

**INSERT INTO** `t`(`column_list`)
VALUES (`value_list`), (`value_list`), ....;
Insert multiple rows into a table

**INSERT INTO** `t`(`column_list`)
**SELECT** `column_list` FROM `t2`;
Insert rows from `t2` into `t`

**UPDATE** `t`;
**SET** `c1` = new_value;
Update new value in the column `c1` for all rows

**UPDATE** `t`;
**SET** `c1` = new_value,
`c2` = new_value
WHERE condition;
Update values in the column `c1`, `c2` that match the condition

**DELETE FROM** `t`;
Delete all data in a table

**DELETE FROM** `t` WHERE condition;
Delete subset of rows in a table

**USING SQL CONSTRAINTS**

**CREATE TABLE** `t`(`
   `c1` INT PRIMARY KEY,
   `c2` INT,
   `FOREIGN KEY` (c2) REFERENCES `t2`(c2)
);
Set `c2` column as a foreign key

**CREATE TABLE** `t`(`
   `c1` INT,
   `c2` INT,
   `UNIQUE`(c2`
`
`);`)
Make the values in `c1` and `c2` unique

**CREATE TABLE** `t`(`
   `c1` INT,
   `c2` INT,
   `CHECK`(c1 > 0 AND c1 >= c2)
);
Ensure c1 > 0 and values in c1 >= c2

**CREATE TABLE** `t`(`
   `c1` INT PRIMARY KEY,
   `c2` VARCHAR NOT NULL
);
Set values in `c2` column not NULL
**SQL Writes**

**UPDATE** Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’

**INSERT INTO** SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99

**DELETE** Product
WHERE price <= 0.99
How?

Example
Game App

DB v0

(Recap lectures)

Q1: 1000 users/sec?
Q2: Offline?
Q3: Support v1, v1’ versions?

Q4: Which user cohorts?
Q5: Next features to build?
Q6: Predict ads demand?
Q7: How to model/evolve game data?
Q8: How to scale to millions of users?
Q9: When machines die, restore game state gracefully?

App designer

Systems designer

Product/Biz designer

Report & Share Business/Product Analysis

Real-Time User Events

DBMS

DB

Mobile Game
How?

Example Game App

DB v0

(Recap lectures)

Q1: 1000 users/sec?
Q2: Offline?
Q3: Support v1, v1' versions?

Q4: Which user cohorts?
Q5: Next features to build?
Q6: Predict ads demand?
Q7: How to model/evolve game data?
Q8: How to scale to millions of users?
Q9: When machines crash, restore game state gracefully?

App designer
Systems designer
Product/Biz designer
1. Why Transactions?

2. Transactions

3. Properties of Transactions: ACID

4. Logging
Example
Unpack ATM DB:
Transaction

Read Balance
Give money
Update Balance
vs
Read Balance
Update Balance
Give money
Visa does > 60,000 TXNs/sec with users & merchants

Want your 4$ Starbucks transaction to wait for a stranger’s 10k$ bet in Las Vegas?
⇒ Transactions can (1) be quick or take a long time, (2) unrelated to you
Transactions are at the core of
-- payment, stock market, banks, ticketing
-- Gmail, Google Docs (e.g., multiple people editing)
Example

Monthly bank interest transaction

Money

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>500</td>
</tr>
<tr>
<td>4001</td>
<td>100</td>
</tr>
<tr>
<td>5001</td>
<td>20</td>
</tr>
<tr>
<td>6001</td>
<td>60</td>
</tr>
<tr>
<td>3002</td>
<td>80</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-100</td>
</tr>
<tr>
<td>40008</td>
<td>100</td>
</tr>
<tr>
<td>50002</td>
<td>20</td>
</tr>
</tbody>
</table>

Money (@4:29 am day+1)

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>550</td>
</tr>
<tr>
<td>4001</td>
<td>110</td>
</tr>
<tr>
<td>5001</td>
<td>22</td>
</tr>
<tr>
<td>6001</td>
<td>66</td>
</tr>
<tr>
<td>3002</td>
<td>88</td>
</tr>
<tr>
<td>4002</td>
<td>-220</td>
</tr>
<tr>
<td>5002</td>
<td>352</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-110</td>
</tr>
<tr>
<td>40008</td>
<td>110</td>
</tr>
<tr>
<td>50002</td>
<td>22</td>
</tr>
</tbody>
</table>

‘T-Monthly-423’
Monthly Interest 10%
4:28 am Starts run on 100M bank accounts
Takes 24 hours to run

UPDATE Money
SET Balance = Balance * 1.1
Example

Monthly bank interest transaction

Performance

Cost to update all data
100M bank accounts → 100M seeks? (worst case)
(@10 msec/seek, that’s 1 million secs)

Problem 1: SLOW :(
Example

Monthly bank interest transaction

With crash

---

'T-Monthly-423'

Monthly Interest 10%

4:28 am Starts run on 100M bank accounts

Takes 24 hours to run

Network outage at 10:29 am,

System access at 10:45 am

---

Money

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>500</td>
</tr>
<tr>
<td>4001</td>
<td>100</td>
</tr>
<tr>
<td>5001</td>
<td>20</td>
</tr>
<tr>
<td>6001</td>
<td>60</td>
</tr>
<tr>
<td>3002</td>
<td>80</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
</tbody>
</table>

Money (@10:45 am)

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>550</td>
</tr>
<tr>
<td>4001</td>
<td>110</td>
</tr>
<tr>
<td>5001</td>
<td>22</td>
</tr>
<tr>
<td>6001</td>
<td>66</td>
</tr>
<tr>
<td>3002</td>
<td>88</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
</tbody>
</table>

---

Did T-Monthly-423 complete?

Which tuples are bad?

Case1: T-Monthly-423 crashed

Case2: T-Monthly-423 completed

4002 deposited 20$ at 10:45 am

Problem 2: Wrong :( 

Primary data structures/algorithms

LOGS

LOCKS

??????

Big Scale

Roadmap
1. Why Transactions?

2. Properties of Transactions: ACID

3. Logging
A transaction (“TXN”) is a sequence of one or more operations (reads or writes) which reflects a single real-world transition.

START TRANSACTION
UPDATE Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’
COMMIT

In the real world, a TXN either happened completely or not at all (e.g., you withdrew 100$ from bank. Or not.)
Transactions in SQL

• In “ad-hoc” SQL, each statement = one transaction

• In a program, multiple statements can be grouped together as a transaction

```
START TRANSACTION
  UPDATE Bank SET amount = amount - 100
  WHERE name = 'Bob'
  UPDATE Bank SET amount = amount + 100
  WHERE name = 'Joe'
COMMIT
```
Motivation for Transactions

Group user actions (reads & writes) into Transactions helps with two goals:

1. **Recovery & Durability**: Keep the data consistent and durable. *Despite system crashes, user canceling TXN part way, etc.*

   **Idea**: Use **LOGS**. Support to “commit” or “rollback” TXNs

2. **Concurrency**: Get better performance by parallelizing TXNs *without* creating ‘bad data.’ *Despite slow disk writes and reads.*

   **Idea**: Use **LOCKS**. Run several user TXNs concurrently.
Example 1: Protection against crashes / aborts

Scenario: Make a CheapProducts table, from a Products table

Client 1:

```
INSERT INTO CheapProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99

DELETE Product
WHERE price <=0.99
```

Crash / abort!

What goes wrong?
Client 1:

```
START TRANSACTION
INSERT INTO CheapProduct(name, price)
   SELECT pname, price
   FROM Product
   WHERE price <= 0.99

DELETE Product
   WHERE price <= 0.99

COMMIT
```

Now we’d be fine! We’ll see how / why this lecture
Example 2: Multiple users: single statements

Client 1: [at 10:01 am]
UPDATE Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’

Client 2: [at 10:01 am]
UPDATE Product
SET Price = Price*0.5
WHERE pname=‘Gizmo’

Two managers attempt to discount products at same time -

What could go wrong?
Client 1: START TRANSACTION
  UPDATE Product
  SET Price = Price – 1.99
  WHERE pname = ‘Gizmo’
  COMMIT

Client 2: START TRANSACTION
  UPDATE Product
  SET Price = Price*0.5
  WHERE pname=‘Gizmo’
  COMMIT

Now works like a charm—we’ll see how/why next lecture…
3. Properties of Transactions
1. Atomicity
2. Consistency
3. Isolation
4. Durability
**ACID: Atomicity**

- TXN is all or nothing
  - *Commits*: all the changes are made
  - *Aborts*: no changes are made
**ACID: Consistency**

- The tables must always satisfy user-specified *integrity constraints*
  - E.g., Account number is unique, Sum of *debits* and of *credits* is 0

- How consistency is achieved:
  - Programmer writes a TXN to go from one consistent state to a consistent state
  - *System* makes sure that the TXN is atomic (e.g., if EXCEPTION, rolls back)
ACID: Isolation

• A TXN executes **concurrently** with other TXNs

• Effect of TXNs is the same as TXNs running one after another

Conceptually,
• similar to OS “sandboxes”
• E.g. TXNs can’t observe each other’s “partial updates”
ACID: Durability

• The effect of a TXN must **persist** after the TXN
  • And after the whole program has terminated
  • And even if there are power failures, crashes, etc.

• \( \Rightarrow \) Write data to durable IO (e.g., disk)
ACID Summary

- **Atomic**
  - State shows either all the effects of TXN, or none of them
- **Consistent**
  - TXN moves from a state where integrity holds, to another where integrity holds
- **Isolated**
  - Effect of TXNs is the same as TXNs running one after another
- **Durable**
  - Once a TXN has committed, its effects remain in the database
A Note: ACID is one popular option!

- Many debates over ACID, both historically and currently
- Some “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs...

⇒ Usually, depends on what consistency and performance your application needs

ACID is an extremely important & successful paradigm, but still debated!
4. Atomicity & Durability via Logging
Conceptual Idea: Trip to Europe

1. Make TODO list. Buy tickets

2. Actual Visit

(Much longer than buying tickets)
Recall (on disks)

- Sequential reads FASTER than random reads
- Sequential writes (aka “appends”) FASTER than random writes

**Big Idea**: LOGs (or log files or ledger)

- Any value that changes? Append to LOG!
  - LOG is a compact “todo” list of data updates
- **Intuition**:
  - Data pages: (a) Update in RAM (fast) (b) Update on disk later (slow)
  - LOGs: (c) Append “todo” in LOGs and (d) control when you Flush LOGs to disk

Many kinds of LOGs. We’ll study a few key ones!
1. How to make/use LOGs?

2. How to make it fast? (Mess with memory and disk)
Basic Idea: (Physical) Logging

Idea:
• Log consists of an ordered list of Update Records
• Log record contains UNDO information for every update!
  <TransactionID, &reference, old value, new value>
  (e.g., key)

What DB does?
• Owns the log “service” for all applications/transactions.
• Appends to log. Flush when necessary — force writes to disk

This is sufficient to UNDO any transaction!
Example

Monthly bank interest transaction

Full run

*T-Monthly-423*
Monthly Interest 10%
4:28 am Starts run on 100M bank accounts
Takes 24 hours to run

```
START TRANSACTION
UPDATE Money
   SET Amt = Amt * 1.10
COMMIT
```
Example
Monthly bank interest transaction
With crash

TXN ‘T-Monthly-423’
Monthly Interest 10%
4:28 am Starts run on 100M bank accounts
Takes 24 hours to run
Network outage at 10:29 am,
System access at 10:45 am

Did T-Monthly-423 complete?
Which tuples are bad?

Case1: T-Monthly-423 was crashed
Case2: T-Monthly-423 completed. 4002 deposited 20$ at 10:45 am

Can you infer from RED log records?
### Example

#### Monthly bank interest transaction

**Recovery**

**Money (@10:45 am)**

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>550</td>
</tr>
<tr>
<td>4001</td>
<td>110</td>
</tr>
<tr>
<td>5001</td>
<td>22</td>
</tr>
<tr>
<td>6001</td>
<td>66</td>
</tr>
<tr>
<td>3002</td>
<td>88</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-110</td>
</tr>
<tr>
<td>40008</td>
<td>110</td>
</tr>
<tr>
<td>50002</td>
<td>22</td>
</tr>
</tbody>
</table>

**Money (after recovery)**

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>500</td>
</tr>
<tr>
<td>4001</td>
<td>100</td>
</tr>
<tr>
<td>5001</td>
<td>20</td>
</tr>
<tr>
<td>6001</td>
<td>60</td>
</tr>
<tr>
<td>3002</td>
<td>80</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-100</td>
</tr>
<tr>
<td>40008</td>
<td>100</td>
</tr>
<tr>
<td>50002</td>
<td>20</td>
</tr>
</tbody>
</table>

**WA Log (@10:29 am)**

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Monthly-423</td>
<td>3001   500   550</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>4001   100   110</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>5001    20    22</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>6001    60    66</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>3002    80    88</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>4002    -200  -200</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>5002    320   320</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>30108   -100  -110</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>40008   100   110</td>
</tr>
<tr>
<td>T-Monthly-423</td>
<td>50002   20    22</td>
</tr>
</tbody>
</table>

**System recovery (after 10:45 am)**

1. Rollback uncommitted transactions
   - Restore old values from WAL Log (if any)
   - Notify developers about aborted TXN
2. Redo Recent transactions (w/ new values)
3. Back in business; Redo (any pending) transactions
1. How to make/use LOGs?

2. ⇒ How to make it fast? (Mess with memory and disk)
A picture of logging

Log is a file (like any data table)
1. Pages updated in RAM
2. Flushed as DB blocks on disk (sequential I/O)

“Flushing to disk” = writing to disk from main memory
A picture of logging

T: R(A=7), W(A=13)
[T reads A=7, writes A=13]

[Update Record]
<Tid, &A, 7,13>

Operation recorded in update log in main memory!
Why do we need logging for atomicity?

• Could we just write TXN updates to disk only once whole TXN complete?
  • Then, if abort / crash and TXN not complete, it has no effect - atomicity!
  • *With unlimited memory and time, this could work…*

• \( \Rightarrow \) We need to log partial results of TXNs because of:
  • Memory constraints (e.g., billions of updates)
  • Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
…And so we need a LOG to (maybe) undo these partial results!
What is the correct way to LOG to disk?

- We’ll look at the Write-Ahead Logging (WAL) protocol
- We’ll see why it works by looking at other protocols which are incorrect!

Remember: Key idea is to ensure durability while maintaining our ability to “undo”!
Write-Ahead Logging (WAL)
TXN Commit Protocol
Write-ahead Logging (WAL) Commit Protocol

Commit after we’ve written log to disk but before we’ve written data to disk…

OK, Commit!

T: R(A), W(A)

Commit record

A=13

B=5

Main Memory

Log-RAM

Log-Disk

Data on Disk

Update Record

<Tid, &A, 7,13>
Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

Commit *after* we’ve written log to disk but *before* we’ve written data to disk… this is WAL!

If we crash now, is T durable?

USE THE LOG!

< Tid, &A, 7, 13 >
Write-Ahead Logging (WAL)

**Algorithm:** WAL

For each tuple update, **write Update Record** into LOG-RAM

Follow two **Flush** rules for LOG

- **Rule1:** Flush **Update Record** into LOG-Disk before corresponding data page goes to storage
- **Rule2:** Before TXN commits,
  - Flush all **Update Records** to LOG-Disk
  - Flush **COMMIT Record** to LOG-Disk

Transaction is committed *once COMMIT record is on stable storage*
Incorrect Commit Protocol #1

Let's try committing before we've written either data or LOG to disk…

If we crash now, is T durable?

OK, Commit!

Lost T’s update!

T: R(A), W(A)

A: 7→13

Main Memory

Log-RAM

A=13

B=5

Data on Disk

A=7

Log-Disk
Incorrect Commit Protocol #2

T: R(A), W(A)  
A: 7→13

A=13  
B=5

Main Memory

Data on Disk

Log-Disk

Log-RAM

Let’s try committing after we’ve written data but before we’ve written LOG to disk…

OK, Commit!

If we crash now, is T durable? Yes! Except…

How do we know whether T was committed??
Example

Monthly bank interest transaction

Performance

### Cost to update all data
100M bank accounts → 100M seeks? (worst case)

(@10 msec/seek, that’s 1 Million secs)

### Cost to Append to log
+ 1 seek to get ‘end of log’
+ write 100M log entries sequentially (fast!!! < 10 sec)

[Lazily update data on disk later, when convenient.]

### Speedup for TXN Commit
1 Million secs vs 10 sec!!!
Logging Summary

• If DB says TX commits, TX effect remains after database crash

• DB can undo actions and help us with atomicity

• This is only half the story...