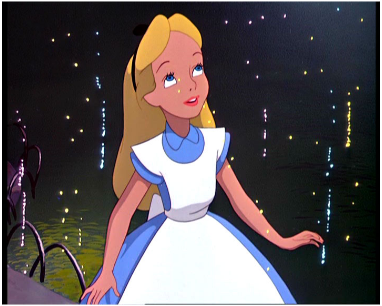
# Fun with Transactions

Joshua Tolley – eggyknap – End Point Corporation

Utah Open Source Conference, 2010

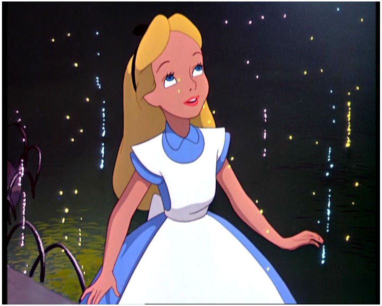
First, an illustration...

# An illustration

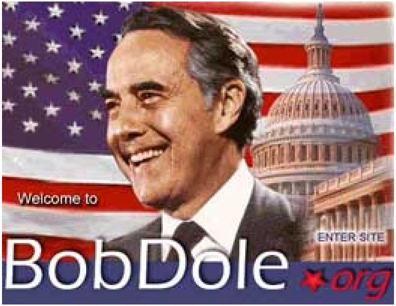


Alice

# An illustration

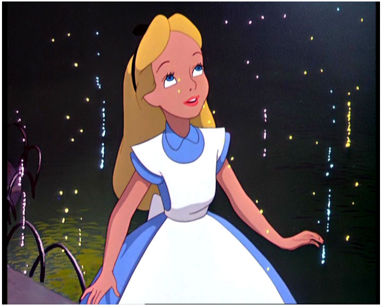


Alice

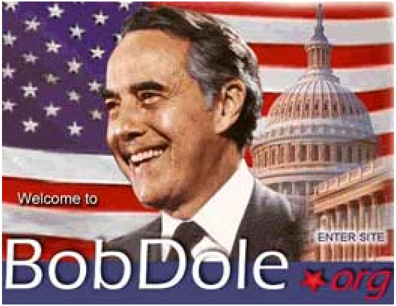



Bob

# An illustration



Alice




Bob

Alice wants to donate $25 to Bob's campaign

# An illustration

Alice gives Bob a check for $25 dollars from East Podunk Savings and Loan. Bob deposits the check.

East Podunk Savings and Loan

# An illustration

East Podunk S & L now has a process they need to follow:

1. Take Alice's check from the pile of checks
2. Debit Alice \$25
3. Credit Bob \$25
4. Mark check as processed
5. Return check to Alice

# An illustration

East Podunk S & L now has a process they need to follow:

1. Take Alice's check from the pile of checks
2. Debit Alice $25
3. Credit Bob $25
4. Mark check as processed
5. Return check to Alice

*What happens when the server crashes?*

# Atomicity

Database designers wanted to specify groups of operations, and have them succeed or fail as a group. This is called **atomicity**.
This gave rise to an API:

- **BEGIN** – Indicate the beginning of a group of operations. These groups are called "transactions"
- **COMMIT** – Perform the operations as an atomic group. It is possible at this point for the commands to fail, and for COMMIT to return an error
- **ROLLBACK** – End this group, and undo the operations of this group

# ACID

Further research developed a list of four characteristics users wanted their transactions to have, which were given the acronym **ACID**:

- **Atomicity** – Groups of operations all either succeed or fail, as a group
- **Consistency** – Users can define constraints on the data. After a transaction finishes, the data set meets each of those constraints.
- **Isolation** – Users can see data only from committed transactions. No one sees my data until I commit it. More on this later
- **Durability** – Once committed, data remain committed, even if the server crashes.

# NoSQL?

In recent years, database users have begun to question the universal application of ACID, such as with the NoSQL movement.

- This is not a bad thing
- Many applications don't need traditional transactions
- Many more applications *do* require traditional transactions, but don't realize it, or are using them incorrectly

True or False: I’m not a bank. Therefore, I don’t need transactions.

True or False: I’m not a bank. Therefore, I don’t need transactions.

*False*

# More examples

One End Point client manages call center data. Their application includes tasks like these:

- Update notes for a support case
- Modify employee performance statistics as technicians take calls
- Update training databases for calls "randomly recorded for training purposes"
- Bill customers for support costs

Failures within any of these operations without transactional guarantees will lead to orphaned records, missing information, and irritated customers.

# More examples

TriSano™(http://www.trisano.org) is a public health reporting application built by an End Point client. For each new case, the application can record some or all of the following:

- Patient demographics
- Participating physicians
- Laboratory tests
- Reports to various agencies
- Contacts of different types
- Custom data collection forms

These data live in many different tables. Changes need to be atomic to prevent orphaned data and ensure consistency.

# ORMs

Object-Relational Mappers (ORMs) often provide APIs for transaction control. Users generally ignore those APIs, and let the ORM follow its default behavior. This behavior is generally wrong.

Transactions group operations logically. Only the programmer knows what groupings are logical. *Ergo*, the programmer (not the ORM, the database system, the database driver, or anything else) should define transactions

- You probably need transactions whether you're ready to admit it or not
- You probably need transactions whether you're ready to write your software to handle it or not
- The application has units of work it needs to keep atomic
- Only the application knows the boundaries of those units
- Unfortunately often your ORM disagrees with these last two

# Neat Transaction Tricks

# Savepoints

Transactions group operations. Within those groups, there may be nested subgroups, called subtransactions, implemented with **savepoints**.

# Savepoint Example

```
BEGIN;
-- Do something useful
INSERT INTO actions VALUES
   ('IM IN YR AKSHUNZ DOIN YOOSFL STUFS');
SAVEPOINT try_something;
SELECT COUNT(*) FROM some_other_table;
UPDATE foo SET bar = baz WHERE qux = 42;
-- Perhaps a constraint causes a failure here
ROLLBACK TO SAVEPOINT try_something;
-- My outer transaction is still in-flight
```

# Implementation note

In PostgreSQL, a statement that generates an error will cause continued errors until a rollback, either to a savepoint or of the transaction entirely. This makes savepoints quite common:

```
josh=# BEGIN;
BEGIN
josh=# SELECT syntax error;
ERROR: syntax error at or near "error"
LINE 1:  SELECT syntax error;
josh=# SELECT 1;
ERROR: current transaction is aborted, commands
ignored until end of transaction block
josh=# ROLLBACK;
```

# Implementation note

MySQL, on the other hand, returns an error message for the syntax error, but still allows the transaction to commit:

```
mysql> start transaction;
mysql> insert into i values (1);
mysql> insert into i values (2);
mysql> insert into i values (1);
ERROR 1062 (23000):  Duplicate entry ’1’ for key 1
mysql> insert into i values (3);
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

# Implementation note

Your mileage may vary. Test, and pay attention to return results and error messages

# Isolation levels

*Isolation* means I can't see values from transaction that's not committed unless it's my own. There are three phenomena it relates to:

- **Dirty read**: A transaction reads data from a concurrent uncommitted transaction
- **Nonrepeatable read**: A transaction re-reads data and finds another transaction has changed them
- **Phantom read**: A transaction re-executes a query, and the set of rows satisfying that query has changed due to some other transaction.

# Isolation levels

The SQL standard defines four isolation levels in terms of these phenomena:

| Isolation Level | Dirty | Non-repeat | Phantom |
|---|---|---|---|
| Read uncommitted | Yes | Yes | Yes |
| Read committed | No | Yes | Yes |
| Repeatable read | No | No | Yes |
| Serializable | No | No | No |

Applications can choose an isolation level appropriate for their needs. Stricter levels may entail poorer performance.

# Beyond the database

A transaction is an atomically committed group of operations. This idea is not limited to databases. Some other examples:

- Message queues
- Integration software
- Transactional memory systems

An operation could, conceivably, want to include multiple transaction-aware services in one transaction:

- Read messages from a queue, and do work in a database in response to those messages
- Use data from multiple separate databases
- Move messages from one messaging service to another
- Build documents using data in a database in a multi-step integration process

This is possible with **distributed transactions**

# Two–phase commit

A *distributed transaction* is a transaction involving multiple services. It works as follows:

1. BEGIN the transaction in each service
2. Perform whatever operations are necessary
3. PREPARE each transaction for commit
   - Each service guarantees at this point that it can commit the transaction, even if something crashes, before returning
4. If any service reports it can't commit, ROLLBACK each transaction
5. If we haven't rolled back, COMMIT each transaction on each service

This is called **two-phase commit** (2PC) because commit happens in two steps

# Two–phase commit

2PC is slower than single-phase commit

- Communicating with $N$ services is slower than communicating with 1 service, where $N > 1$.
- On PREPARE, each transaction must store transaction state on disk, to ensure durability
- The application must also store its own state on disk, in case it crashes between PREPARE and COMMIT
  - Applications generally use *transaction managers* rather than handle these details on their own

# Bitronix transaction manager example

```
#!/bin/jruby
require 'java'

BTM = Java::BitronixTm::bitronixTransactionManager
TxnSvc = Java::BitronixTm::TransactionManagerServices
PDS = Java::BitronixTmResourceJdbc::PoolingDataSource

# Do the stuff below for each data source
ds1 = PDS.new
ds1.set_class_name 'org.postgresql.xa.PGXADataSource'
...
ds1.init
# Get datasource connections, and start a transaction
c1 = ds1.get_connection
c2 = ds2.get_connection
btm = TxnSvc.get_transaction_manager
btm.begin
```

# Bitronix transaction manager example

```
begin
        # Do something on each connection
        s2 = c2.prepare_statement
               "INSERT INTO ledger VALUES ('Bob', 100)"
        s2.execute_update
        s2.close
        ...
        btm.commit
        puts "Successfully committed"
rescue
        puts "Something bad happened:  " + $!
        btm.rollback
end
```

# Transaction pitfalls

# Stuff to watch out for

- Long transactions are typically bad
    - Locks remain held until transactions commit
    - Rollback space can't be released until commit
    - This may be particularly bad with 2PC
- More complex transactions are probably more likely to roll back
- Handle exceptions properly (you're doing this already, right?)
    - If you roll back and try again, you have to redo the entire operation

# Transaction benefits

# Miscellaneous. transaction benefits

- Performance
  - Several INSERTs with Auto–commit will generally be much slower than the same inserts in a single transaction
- Simpler code
  - Wrap one transaction in one exception handling block
- Data integrity
  - Orphaned records and other data anomalies are much less likely

**Questions?**

# Fun with Transactions

Joshua Tolley – eggyknap – End Point Corporation

Utah Open Source Conference, 2010

# Fun with Transactions

Joshua Tolley – eggyknap – End Point Corporation

Utah Open Source Conference, 2010