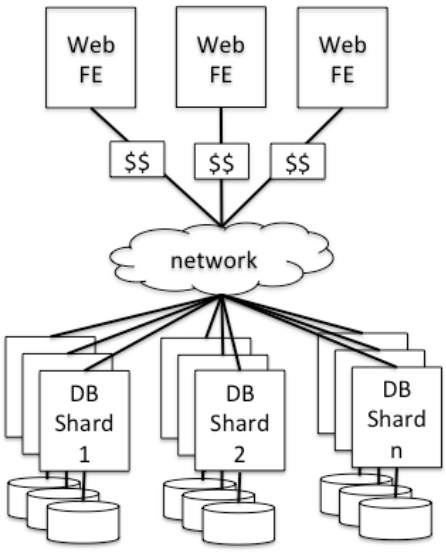
COMMITMENT PROTOCOLS ON SHARDED DATABASES

We talked about sharding as a key mechanism for scaling distributed systems. We also talked about replication as a key mechanism for fault tolerance in distributed systems. Both challenge coordination and raise semantic challenges. To handle the challenges, the DS community has come up with rigorous protocols for coordination. The protocols differ depending on the type of distributed system you are developing – e.g., you'll have different protocols for dealing with coordination in distributed databases vs. distributed computation engines. In this module we will focus on the specific case of distributed databases. Over the next three lectures we'll look at protocols for dealing with semantic challenges raised by sharding and replication in distributed systems. The next two lectures look at either sharding or replication separately; the third lecture shows how to combine the techniques to address the joint challenges occurring in real distributed databases, which are both sharded and replicated.

So: for this lecture, I want you to forget about replication and fault tolerance, and focus instead on sharding and scalability.

I. EXAMPLE SEMANTIC CHALLENGES WITH SHARDING

Going back to the Web service example we presented in the previous lecture, let's revisit the last architecture we discussed, but ignore the replication aspect of it. For concreteness, let's specify what the application (implemented by the FE) is in this case: say it is a banking application, where user account information is stored in the DB and users can perform money transfer transactions to move data from one account to another as long as they have sufficient funds in the source account.



Architecture: FE and DB are both sharded. FEs accept requests from end-users' browsers and process them concurrently (i.e., two requests from the same or from different users can execute in parallel on two different FEs). The data stored in the DB is sharded, say by user ID. All data of the first third of the users in ID space is stored in Shard 1; all data of the next third of the users in ID space is stored on Shard 2; and all data of the last third of the users in ID space is stored on Shard n.

First question: How does this architecture increase the capacity of your system (i.e., the maximum load the system can handle)?
Answer: As long as most of the workload involves accessing accounts from one of the shards (e.g., users check out mostly their accounts), the overall capacity should increase, because you now have the various DB shards handling data requests in parallel from multiple different FEs.

Second question: What semantics challenges does this architecture raise?
Answer: Many, here's an example. Suppose a user wants to transfer some money from his account to another user's account. This bank "transaction" involves two operations on two different parts of the database – deducting the money from the source account and adding it to the destination account. Sometimes, these two accounts will be stored on different machines. From a semantic perspective, it's

important that both operations either succeed or fail, otherwise you can either "lose" money (if the operation completes on the source but not on the destination) or "create" money out of thin air (if the operation fails on the source but completes on the destination). Unfortunately, the two machines are independent and hence they can fail independently, so what we would like is a coordination protocol between the two DB shard servers that lets us ensure that the two operations can either both succeed, or if one fails, the other one is not applied, either.

That is what agreement protocols give you, and the best known agreement protocol is Two-Phase Commit (2PC). We'll talk about it in this lecture, but first have to introduce some more basic notions, namely the concept of transactions, which is the abstraction that databases offer to support these kinds of multi-operation needs. We'll then look at how transactions are implemented in a single-node, non-sharded case, and then at the end of the lecture we'll look at 2PC, which makes transactions work in the distributed/sharded case.

II. TRANSACTIONS – THE CONCEPT

A Turing-award-winning idea; a transaction is an abstraction provided to programmers that **encapsulates a unit of work against a database**. Transactions provide a simple but powerful interface:

- txID = **begin**() // starts a transaction; returns a unique ID for the transaction
- outcome = **commit**(txID) // tries to commit a transaction; returns whether or not the commit
  // was successful. If successful, all operations included in the transaction have
  // been applied to the DB. If unsuccessful, none of the operations have been
  // applied.
- **abort**(txID) // cancels all operations of a transaction and erases their effects on the DB. Can
  // be called by the programmer or by the database engine itself.

By wrapping a set of accesses and updates in a transaction, the database guarantees:

- **Atomicity:** Either all operations in the transaction will complete successfully (commit outcome), or none of them will (abort outcome).
  - Said differently, after a transaction commits or aborts, the database will not reflect a partial result of that transaction.
  - All transactions will either commit or abort.
  - Q: if one were to guarantee failure-freeness, does atomicity come "for free"?
    - A: yes, though it is a wide definition of "failure" for this to be true, e.g., no rollback of conflicting or deadlocking transactions.
- **Isolation**: A transaction's behavior is not impacted by the presence of other, concurrently executing transactions.
  - Said differently, a transaction will "see" only the state of the DB that would occur if the transaction were the only one running against the database, and it will produce results that it could produce if it were running alone.
  - Q: if one executes only a single transaction at a time, does "isolation" come for free?
    - A: yes! this is tied to the very definition of isolation.
- **Durability**: The effects of committed transactions survive failures.
  - If there is non-volatile storage in the system: the effects of a committed transaction must be reflected in non-volatile storage at all times.

- After a failure, the effects of committed transactions must be recoverable or already reflected in the DB.

These properties are often called **ACID** (yes, there's a C that stands for Consistency, but we don't discuss it in this course).

To illustrate transactions, let's go back to the banking example, but let's focus on the single-node/non-sharded case. And let's assume that there is concurrency: the DB is multi-threaded and it's processing transactions from multiple clients in parallel. There are challenges with how one implements transactions even in this case. ((Note that although we are using the banking example, the notion of transactions is much more vastly applicable, even to applications that don't seem to truly "need" strong transactional semantics. The reason is that it's a lot simpler to program against a strong-semantic database system than it is to program against a weaker-semantic one.)) Here's a piece of code for two programs running on top of a database that offers transactions as a programming abstraction.

```
TRANSFER(src, dst, x)
01   src_bal = Read(src)
02   if (src_bal > x):
03      src_bal -= x
04      Write(src_bal, src)
05      dst_bal = Read(dst)
06      dst_bal += x
07      Write(dst_bal, dst)

Invocation: TRANSFER(A, B, 50)
```

```
REPORT_SUM(acc1, acc2)
01   acc1_bal = Read(acc1)
02   acc2_bal = Read(acc2)
03   Print(acc1_bal + acc2_bal)

Invocation: PRINT_SUM(A, B)
```

Without transactions: What could go wrong? Think of crashes or inopportune interleavings between concurrent TRANSFER and REPORT_SUM processes .

TRANSFER:

- Assume statements operates on DB immediately, and DB is a single data structure
  - what happens if there is a crash after 04 but before 07?
  - money is lost
- Why? Because the transfer is **not atomic**
  - need some way of making sure entire transfer happens, or none.
  - That's what the **atomicity property** of ACID transactions gives.

REPORT_SUM:

- Fine if ReportSum() executes before or after TRANSFER()
- what happens if interleaved with TRANSFER()?
  - Depends on the interleaving, some are OK, others not. The following is not OK.
  - Suppose REPORT_SUM's steps are all interleaved between TRANSFER's steps 04 and 05. It will seem like some money was lost. Why is that not OK (for the example)? Suppose A and B are joint accounts, and one owner is transferring money and at the same time another owner checks for the total balance, then the latter will become very confused and will think they've lost some money.

    - Why is that not OK more generally, beyond this trivialized example?
        - Because REPORT_SUM and TRANSFER both depend on the same data
        - And TRANSFER is modifying that data
        - And REPORT_SUM sees both data that predates TRANSFER() (B) and post-dates TRANSFER() (A)
        - That violates this illusion of sequential execution! We lack **isolation**.
        - And the big problem may not be that users get confused, but even worse, applications may get confused, and if they aren't coded to take into account all of these corner-case situations that may happen, they may fail. And it's really, really hard to think about all corner cases. That's why we want strong semantics, so we (as programmers) don't have to worry about corner cases.
- That's what the **isolation** property of ACID transactions give.

With transactions: To fix these challenges, you just modify the TRANSFER and REPORT_SUM to wrap their operations into a transaction, i.e., add begin() and commit() at the beginning and end, respectively, of each method.

So, the idea is that if you build your applications on top of ACID transactions, you won't have to worry about the challenges we described. By and large, there are things to pay for using these strong semantic: transactions *are* expensive, so sometimes you may have to forego their strongest semantics and make do with something weaker. So it's good to understand a bit how these strong semantics are implemented so you can reason about their costs in your application, and potentially how you can weaken them in a way that is still meaningful to your application but more efficient. So, in the next section, we'll look at how single-node, non-sharded databases implement ACID transactions. Section IV talks about how to implement transactions in the sharded-database case.

III. IMPLEMENTING TRANSACTIONS IN A SINGLE-NODE, NON-SHARDED DB

Based on the preceding examples, we need to address two challenges to implement ACID transactions even in a *non-distributed database*.

Atomicity and durability challenges: How do we make sure that the operations included in a transaction either all succeed or none of them succeed despite temporary failures of the machine running the DB? (Remember no distribution in this section.) The key mechanism here is **write-ahead logging**. Assume that disks are reliable and cannot fail, but that machines can fail temporarily. Upon recovery, they can access the data on disk but RAM data is vanished. The idea in write-ahead logging is to log to disk sufficient information about each operation *before you apply it to the database*, such that in the event of a failure in the middle of a transaction, you can undo the effects of its operations on the database. If you've managed to apply all the operations in a transaction without a failure, then you enter in your log that the particular transaction is completed. Upon a subsequent failure of the DB server, the server will read the logs and apply all transactions that are committed and undo any transactions that were still ongoing at the time of failure. These slides <https://columbia.github.io/ds1-class/lectures/09-local-transactions-wal.pdf>, courtesy of Dave Andersen, describe this mechanism with an example.

With write-ahead logging, you get the following semantic: (1) the operations in a transaction are completed as a unit, i.e., either all (commit outcome) or none (abort outcome) and (2) for any committed

transaction, its effects will persist despite database failures, and become available after recovery. The two parts of the semantic correspond to atomicity and durability properties of ACID.

Isolation challenge: How do we make sure that the operations included in a transaction all witness the database in a coherent state, independent of other ongoing (a.k.a., concurrent) transactions? The key mechanism here is **locking**. In one instantiation of this mechanism, the DB acquires locks on all rows read or written and maintains them until the end of the transaction. Read but not written rows can be locked in a shared way, allowing other transactions to read (but not write) them. Rows that are written are locked exclusively (no other readers/writers allowed). There are some challenges that one needs to worry about with locks, including deadlocks; if you are interested, you can refer to these slides <https://columbia.github.io/ds1-class/lectures/09-local-transactions-2pl.pdf> (also courtesy of Dave Andersen), but they are not required for exam (unlike the WAL slides and other slides linked from the class notes).

Databases use two-phase locking to achieve various levels of isolation between concurrent transactions: If the DB grabs locks for both read and written rows, and retain them till the end of the transaction, then you can get very strong isolation semantics (called *serializability*), but that can be very expensive. This is because other transactions that are trying to access some of the rows you've locked will be blocked waiting for your transaction to finish. Instead, if the DB grabs only locks for written rows, and holds each only while it performs each write operation, then you get weaker isolation semantics (called *read uncommitted*), but that mechanism allows for greater concurrency and hence it's more efficient. In between these two semantics, there are other semantics of intermediary strength and overhead.

In your applications, you will have to choose the strength of the semantic, so it's good to familiarize yourselves with each semantic before you make a choice. In general, the idea is that stronger semantic means you'll find it easier and more intuitive to build your application, but you'll sacrifice performance. Weak semantic means that you'll have to code your application around weird corner cases that the semantic allows, so it's harder to code but it can be made faster.

IV. IMPLEMENTING TRANSACTIONS IN A SHARDED DB

Now that we understand what transactions are, and how they are implemented in the single-node DB case, let's see how they are implemented in the sharded case. What the single-node case gives us is a way to implement the various operations included in a transaction, in an atomic and isolated way, against one of the nodes. In the sharded case, a transaction will consist of several operations, subgroups of which will be executed against different servers. We would like to preserve the same atomicity, durability, and isolation semantics as we did in the single-node case, but in a sharded case now. What we'll do is to execute each portion of the transaction that is relevant for each shard using the techniques from Section III, and then we'll devise a protocol for the various shards involved in a transaction to decide whether they should all commit their portions of the transaction or not.

The protocol is called **two-phase commit**, and we'll first describe it in the context of the example in Section II. After that we'll generalize.

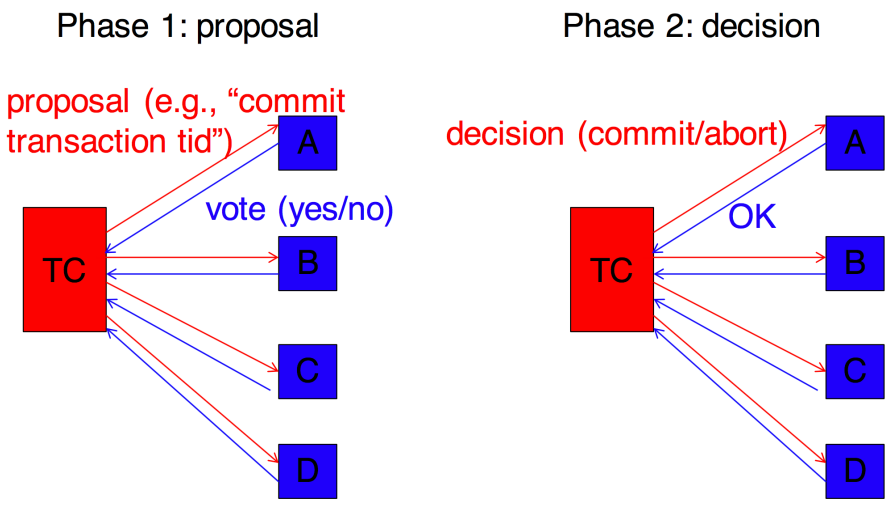
Simplified 2PC based on Section II Example

Suppose you have a simple transaction, which consists of two operations only: deduct money from the source account account (op1) and add it to a destination account (op2). And suppose the source and destination accounts are stored on different servers (due to DB sharding), S1 and S2, respectively. The operation relevant to S1 will be op1; the operation relevant to S2 will be op2. We would like either both or neither of these operations to be executed on the two servers. We'll do several things:

1. To begin a distributed transaction, the client (in this case one of the FEs, on behalf of the end user) initiates transactions on each separate shard server.
2. As part of the distributed transaction, the client will send the operation to the corresponding shard server. Op1 goes to S1 and op2 goes to S2. Each shard will perform these operations as we described in the previous section: S1 will lock the row for the source account, will log sufficient information about op1 to its write-ahead log to be able to undo op1 if necessary, and finally will perform the operation on the database; S2 will do similarly for op2.
3. When it's time to **commit** the transaction, we will proceed as follows:
   3.1. The client (or rather, the DB library on behalf of the client code) sends a **PREPARE** message to each shard server, specifying the transaction ID.
   3.2. Upon receipt of a **PREPARE** message for a transaction txID, a shard server, Si, will: (1) grab the locks that are necessary to fulfill that transaction (in this case, S1 will grab the lock for the src account row, and S2 will grab the lock fort he destination account row); (2) while holding the locks, will write each operation to the write-ahead log, including sufficient information for a potential undo later on; (3) updates the database (S1 will update the balance for the source account and S2 will update the balance for the destination account); and (4) reply **OK** to the client if it has successfully completed steps (1)-(3) (i.e., it hasn't failed in the meantime). If instead, the server has failed somewhere in the meantime, or for other reasons isn't able to commit the transaction (e.g., S1 might find that there are insufficient funds in the source account to make the transaction), then the server will send a **FAIL** response back to the client and will proceed to undo the transaction based on its write-ahead log and release all of its locks. Note that if the server sent an **OK** in Step (4), then the node has to wait for a next message from the client before it releases its locks and marks the transaction as committed in the write-ahead log.
   3.3. On the client, upon receiving OK/FAIL responses from the shard servers:
      3.3.1. If the client receives **OKs from both servers**, then the client sends a **COMMIT** message to both servers S1 and S2. It then waits again for the acknowledgement.
      3.3.2. If the client receives **one FAIL response**, then the client will send an **ABORT** message to both servers S1 and S2.
   3.4. Upon receipt of a **COMMIT** or **ABORT** message from the client, a shard server, Si, will: (1) enter commit/abort in its write-ahead log, (2) if it's ABORT then it goes on and reverts the effects of the transaction on the database, and (3) release the locks it was holding for the transaction.

More General Version of the Protocol

The protocol we presented above is driven by our example and is a simplification of the actual 2PC protocol. More generally, two 2PC protocol is performed across two or more nodes (called *participants)*, and in the picture below, which illustrates the more general version, it is performed across four participants. Also, in more general settings, it is a really bad idea to have the client – an external entity w.r.t. the DB service – coordinate the transaction. Instead, it helps if one has a designated server, called a *transaction coordinator*, send the PREPARE (aka, proposal) and COMMIT (aka, decision) messages. The coordinator can then record the various phases at the protocol into its own log to help with recovery from various conditions (we'll discuss recovery next). So, in the picture below, we show the two phases of the 2PC protocol, each corresponding to a message exchange (prepare and commit), and a transaction coordinator (denoted TC).



Handling Timeouts/Failures

The preceding protocol description deals with "happy cases," but doesn't specify what happens on various failures. Let's study those situations next.

Timeouts:

- waits are before steps 2, 3, and 4.
- 2: before participant has voted, so safe to abort on timeout
- 3: client is waiting for OK or FAIL from participants; so, safe to abort on timeout
- 4: the only **uncertainty period**. After timeout, need a **termination protocol**:
    a) "wait until communication with coordinator is re-established." Safe but downside is p may be blocked unnecessarily, since if it can learn the decision from any other participant that has decided.
    b) "cooperative" -- participants know about each other, and p pings q for outcome. (if q is not in uncertainty period and has not decided, q can abort as the outcome!)

Recovery:

- if participant is not in uncertainty period, on recovery, can decide what to do. (unilaterally abort if no decision, otherwise do what decision is.)
- if participant is in uncertainty period, it cannot decide on its own, must run **termination protocol** (as above).

V. CONCLUSION

Sharding creates an atomicity challenge. To address it, we introduced a common and useful abstraction in (distributed) databases: **transactions**, which represent atomic units of work. We've looked at how transactions are implemented in a single-database case – which is difficult in it of itself with strong semantics – and then we looked at two-phase commit, a commitment protocol for implementing transactions in a distributed, sharded database. Recall that the whole purpose we were applying sharding on the database was to be able to add more servers to increase the capacity of the database (i.e., be able to handle more concurrent requests). However, with transactions, where two or more shards need to coordinate tightly (and hold locks while doing so!), the capacity gain may not be that great. I.e., you will certainly not get, with three shard servers, three times the capacity of each server. The gain you get will very much depend on the workload and how effectively sharding separates transactions: if most transactions interact with only one shard, you'll gain a lot of capacity; if most transactions use many shards, you'll gain nothing, you may actually have worse capacity than if you weren't sharding. So think carefully before you decide to use distribution as a method for increasing your capacity.

Next time we will see how to add fault tolerance into the system through replication, and then how to address the semantic challenges raised by this replication.