In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability.

Web applications have grown in popularity over the past decade. Whether you are building an application for end users or application developers (i.e., services), your hope is most likely that your application will find broad adoption—and with broad adoption will come transactional growth. If your application relies upon persistence, then data storage will probably become your bottleneck.

There are two strategies for scaling any application. The first, and by far the easiest, is vertical scaling: moving the application to larger computers. Vertical scaling works reasonably well for data but has several limitations. The most obvious limitation is outgrowing the capacity of the largest system available. Vertical scaling is also expensive, as adding transactional capacity usually requires purchasing the next larger system. Vertical scaling often creates vendor lock, further adding to costs.

Horizontal scaling offers more flexibility but is also considerably more complex. Horizontal data scaling can be performed along two vectors. Functional scaling involves grouping data by function and spreading func-

BA

DAN PRITCHETT, EBAY

tional groups across databases. Splitting data within functional areas across multiple databases, or *sharding*,[1] adds the second dimension to horizontal scaling. The diagram in figure 1 illustrates horizontal data-scaling strategies.

As figure 1 illustrates, both approaches to horizontal scaling can be applied at once. Users, products, and transactions can be in separate databases. Additionally, each functional area can be split across multiple databases for transactional capacity. As shown in the diagram, functional areas can be scaled independently of one another.

FUNCTIONAL PARTITIONING

Functional partitioning is important for achieving high degrees of scalability. Any good database architecture will decompose the schema into tables grouped by functionality. Users, products, transactions, and communication are examples of functional areas. Leveraging database concepts such as foreign keys is a common approach for maintaining consistency across these functional areas.

Relying on database constraints to ensure consistency across functional groups creates a coupling of the schema

BASE
AN ACID ALTERNATIVE

to a database deployment strategy. For constraints to be applied, the tables must reside on a single database server, precluding horizontal scaling as transaction rates grow. In many cases, the easiest scale-out opportunity is moving functional groups of data onto discrete database servers.

Schemas that can scale to very high transaction volumes will place functionally distinct data on different database servers. This requires moving data constraints out of the database and into the application. This also introduces several challenges that are addressed later in this article.

CAP THEOREM

Eric Brewer, a professor at the University of California, Berkeley, and cofounder and chief scientist at Inktomi, made the conjecture that Web services cannot ensure all three of the following properties at once (signified by the acronym CAP):[2]

Consistency. The client perceives that a set of operations has occurred all at once.

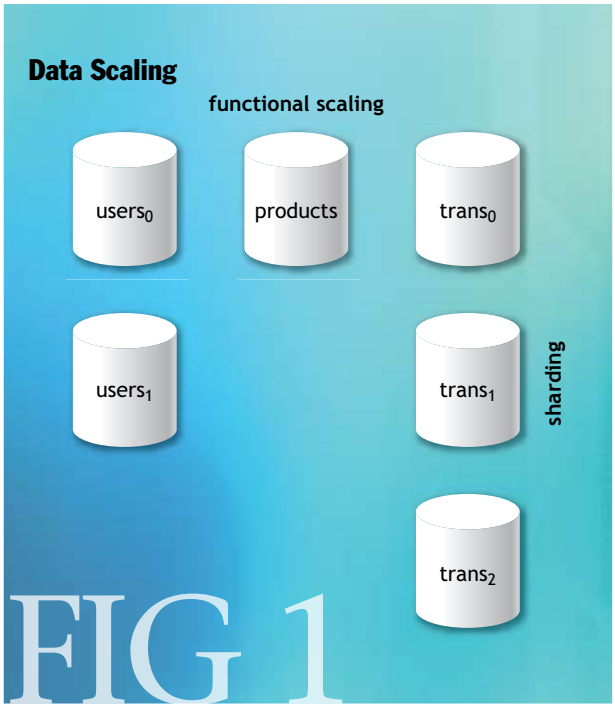


FIG 1

Availability. Every operation must terminate in an intended response.

Partition tolerance. Operations will complete, even if individual components are unavailable.

Specifically, a Web application can support, at most, only two of these properties with any database design. Obviously, any horizontal scaling strategy is based on data partitioning; therefore, designers are forced to decide between consistency and availability.

ACID SOLUTIONS

ACID database transactions greatly simplify the job of the application developer. As signified by the acronym, ACID transactions provide the following guarantees:

Atomicity. All of the operations in the transaction will complete, or none will.

Consistency. The database will be in a consistent state when the transaction begins and ends.

Isolation. The transaction will behave as if it is the only operation being performed upon the database.

Durability. Upon completion of the transaction, the operation will not be reversed.

Database vendors long ago recognized the need for partitioning databases and introduced a technique known as 2PC (two-phase commit) for providing ACID guarantees across multiple database instances. The protocol is broken into two phases:

- First, the transaction coordinator asks each database involved to precommit the operation and indicate whether commit is possible. If all databases agree the commit can proceed, then phase 2 begins.
- The transaction coordinator asks each database to commit the data.

If any database vetoes the commit, then all databases are asked to roll back their portions of the transaction.

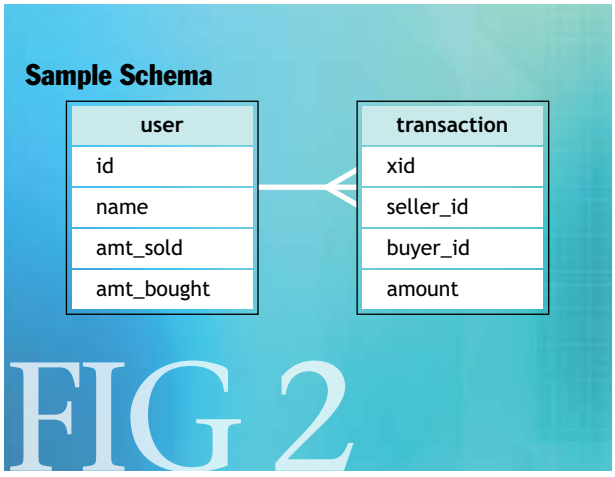


FIG 2

What is the shortcoming? We are getting consistency across partitions. If Brewer is correct, then we must be impacting availability, but how can that be?

The availability of any system is the product of the availability of the components required for operation. The last part of that statement is the most important. Components that may be used by the system but are not required do not reduce system availability. A transaction involving two databases in a 2PC commit will have the availability of the product of the availability of each database. For example, if we assume each database has 99.9 percent availability, then the availability of the transaction becomes 99.8 percent, or an additional downtime of 43 minutes per month.

AN ACID ALTERNATIVE

If ACID provides the consistency choice for partitioned databases, then how do you achieve availability instead? One answer is BASE (basically available, soft state, eventually consistent).

BASE is diametrically opposed to ACID. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this sounds impossible to cope with, in reality it is quite manageable and leads to levels of scalability that cannot be obtained with ACID.

The availability of BASE is achieved through supporting partial failures without total system failure. Here is a simple example: if users are partitioned across five database servers, BASE design encourages crafting operations in such a way that a user database failure impacts only the 20 percent of the users on that particular host. There is no magic involved, but this does lead to higher perceived availability of the system.

So, now that you have decomposed your data into functional groups and partitioned the busiest groups across multiple databases, how do you incorporate BASE into your application? BASE requires a more in-depth analysis of the operations within a logical transaction than is typically applied to ACID. What should you be looking for? The following sections provide some direction.

CONSISTENCY PATTERNS

Following Brewer's conjecture, if BASE allows for availability in a partitioned database, then opportunities to relax consistency have to be identified. This is often difficult because the tendency of both business stakeholders and developers is to assert that consistency is paramount to the success of the application. Temporal inconsistency cannot be hidden from the end user, so both engineering and product owners must be involved in picking the opportunities for relaxing consistency.

Figure 2 is a simple schema that illustrates consistency considerations for BASE. The user table holds user information including the total amount sold and bought. These are running totals. The transaction table holds each transaction, relating the seller and buyer and the amount of the transaction. These are gross oversimplifications of real tables but contain the necessary elements for illustrating several aspects of consistency.

In general, consistency *across* functional groups is easier to relax than *within* functional groups. The example schema has two functional groups: users and transactions. Each time an item is sold, a row is added to the transaction table and the counters for the buyer and seller

```
Begin transaction
  Insert into transaction(xid, seller_id, buyer_id, amount);
  Update user set amt_sold=amt_sold+$amount where id=$seller_id;
  Update user set amt_bought=amount_bought+$amount where id=$buyer_id;
End transaction
```

FIG 3

are updated. Using an ACID-style transaction, the SQL would be as shown in figure 3.

The total bought and sold columns in the user table can be considered a cache of the transaction table. It is present for efficiency of the system. Given this, the constraint on consistency could be relaxed. The buyer and seller expectations can be set so their running balances do not reflect the result of a transaction immediately. This is not uncommon, and in fact people encounter this delay between a transaction and their running balance regularly (e.g., ATM withdrawals and cellphone calls).

How the SQL statements are modified to relax consistency depends upon how the running balances are defined. If they are simply estimates, meaning that some transactions can be missed, the changes are quite simple, as shown in figure 4.

We've now decoupled the updates to the user and transaction tables. Consistency between the tables is not guaranteed. In fact, a failure between the first and second transaction will result in the user table being permanently inconsistent, but if the contract stipulates that the running totals are estimates, this may be adequate.

What if estimates are not acceptable, though? How can you still decouple the user and transaction updates? Introducing a persistent message queue solves the problem. There are several choices for implementing persistent messages. The most critical factor in implementing the queue, however, is ensuring that the backing persistence is on the same resource as the database. This is necessary to allow the queue to be transactionally committed without involving a 2PC. Now the SQL operations look a bit different, as shown in figure 5.

```
Begin transaction
  Insert into transaction(id, seller_id, buyer_id, amount);
End transaction
Begin transaction
  Update user set amt_sold=amt_sold+$amount where id=$seller_id;
  Update user set amt_bought=amount_bought+$amount
        where id=$buyer_id;
End transaction
```

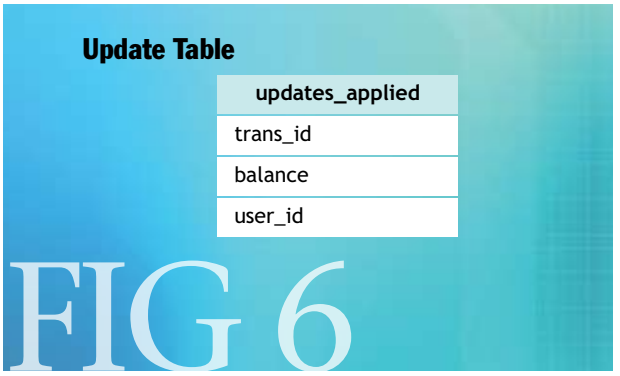
FIG 4

```
Begin transaction
  Insert into transaction(id, seller_id, buyer_id, amount);
  Queue message "update user("seller", seller_id, amount)";
  Queue message "update user("buyer", buyer_id, amount)";
End transaction
For each message in queue
  Begin transaction
    Dequeue message
    If message.balance == "seller"
      Update user set amt_sold=amt_sold + message.amount
            where id=message.id;
    Else
      Update user set amt_bought=amt_bought + message.amount
            where id=message.id;
    End if
  End transaction
End for
```

FIG 5

This example takes some liberties with syntax and oversimplifying the logic to illustrate the concept. By queuing a persistent message within the same transaction as the insert, the information needed to update the running balances on the user has been captured. The transaction is contained on a single database instance and therefore will not impact system availability.

A separate message-processing component will



dequeue each message and apply the information to the user table. The example appears to solve all of the issues, but there is a problem. The message persistence is on the transaction host to avoid a 2PC during queuing. If the message is dequeued inside a transaction involving the user host, we still have a 2PC situation.

One solution to the 2PC in the message-processing component is to do nothing. By decoupling the update into a separate back-end component, you preserve the availability of your customer-facing component. The lower availability of the message processor may be acceptable for business requirements.

Suppose, however, that 2PC is simply never acceptable in your system. How can this problem be solved? First, you need to understand the concept of *idempotence*. An operation is considered idempotent if it can be applied one time or multiple times with the same result. Idempotent operations are useful in that they permit partial failures, as applying them repeatedly does not change the final state of the system.

The selected example is problematic when looking for idempotence. Update operations are rarely idempotent. The example increments balance columns in place. Applying this operation more than once obviously will result in an incorrect balance. Even update operations that simply set a value, however, are not idempotent with regard to order of operations. If the system cannot guarantee that updates will be applied in the order they are received, the final state of the system will be incorrect. More on this later.

In the case of balance updates, you need a way to track which updates have been applied successfully and which are still outstanding. One technique is to use a table that records the transaction identifiers that have been applied.

The table shown in figure 6 tracks the transaction ID, which balance has been updated, and the user ID where the balance was applied. Now our sample pseudocode is as shown in figure 7.

```
Begin transaction
  Insert into transaction(id, seller_id, buyer_id, amount);
  Queue message "update user("seller", seller_id, amount)";
  Queue message "update user("buyer", buyer_id, amount)";
End transaction
For each message in queue
  Peek message
  Begin transaction
    Select count(*) as processed where trans_id=message.trans_id
       and balance=message.balance and user_id=message.user_id
    If processed == 0
      If message.balance == "seller"
        Update user set amt_sold=amt_sold + message.amount
              where id=message.id;
      Else
        Update user set amt_bought=amt_bought + message.amount
              where id=message.id;
      End if
      Insert into updates_applied
        (message.trans_id, message.balance, message.user_id);
    End if
  End transaction
  If transaction successful
    Remove message from queue
  End if
End for
```

FIG 7

This example depends upon being able to peek a message in the queue and remove it once successfully processed. This can be done with two independent transactions if necessary: one on the message queue and one on the user database. Queue operations are not committed unless

database operations successfully commit. The algorithm now supports partial failures and still provides transactional guarantees without resorting to 2PC.

There is a simpler technique for assuring idempotent updates if the only concern is ordering. Let's change our sample schema just a bit to illustrate the challenge and the solution (see figure 8). Suppose you also want to track the last date of sale and purchase for the user. You can rely on a similar scheme of updating the date with a message, but there is one problem.

Suppose two purchases occur within a short time window, and our message system doesn't ensure ordered operations. You now have a situation where, depending upon which order the messages are processed in, you will have an incorrect value for last_purchase. Fortunately, this kind of update can be handled with a minor modification to the SQL, as illustrated in figure 9.

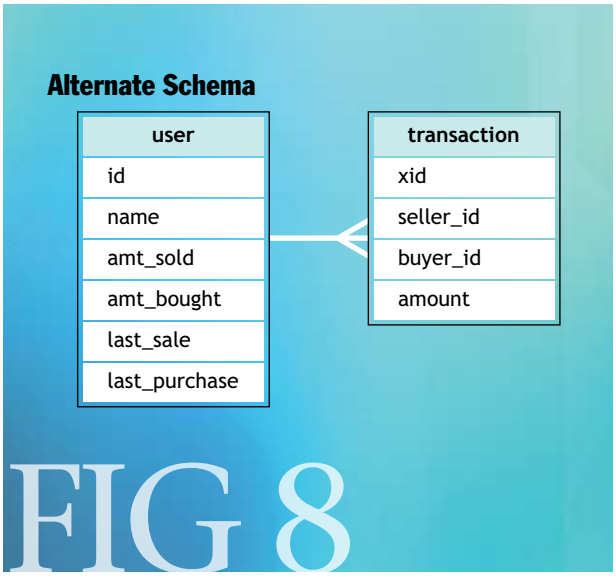


FIG 8

By simply not allowing the last_purchase time to go backward in time, you have made the update operations order independent. You can also use this approach to protect any update from out-of-order updates. As an alternative to using time, you can also try a monotonically increasing transaction ID.

ORDERING OF MESSAGE QUEUES

A short side note on ordered message delivery is relevant. Message systems offer the ability to ensure that messages are delivered in the order they are received. This can be expensive to support and is often unnecessary, and, in fact, at times gives a false sense of security.

The examples provided here illustrate how message ordering can be relaxed and still provide a consistent view of the database, eventually. The overhead required to relax the ordering is nominal and in most cases is significantly less than enforcing ordering in the message system.

Further, a Web application is semantically an event-driven system regardless of the style of interaction. The client requests arrive to the system in arbitrary order. Processing time required per request varies. Request scheduling throughout the components of the systems is nondeterministic, resulting in nondeterministic queuing of messages. Requiring the order to be preserved gives a false sense of security. The simple reality is that nondeterministic inputs will lead to nondeterministic outputs.

SOFT STATE/EVENTUALLY CONSISTENT

Up to this point, the focus has been on trading consistency for availability. The other side of the coin is understanding the influence that soft state and eventual consistency has on application design.

As software engineers we tend to look at our systems

```
For each message in queue:
  Peek message
  Begin transaction
    Update user set last_purchase=message.trans_date where id=message.buyer_id and last_purchase<message.trans_date;
  End transaction
  If transaction successful
    Remove message from queue
End for
```

FIG 9

as closed loops. We think about the predictability of their behavior in terms of predictable inputs producing predictable outputs. This is a necessity for creating correct software systems. The good news in many cases is that using BASE doesn't change the predictability of a system as a closed loop, but it does require looking at the behavior in total.

A simple example can help illustrate the point. Consider a system where users can transfer assets to other users. The type of asset is irrelevant—it could be money or objects in a game. For this example, we will assume that we have decoupled the two operations of taking the asset from one user and giving it to the other with a message queue used to provide the decoupling.

Immediately, this system feels nondeterministic and problematic. There is a period of time where the asset has left one user and has not arrived at the other. The size of this time window can be determined by the messaging system design. Regardless, there is a lag between the begin and end states where neither user appears to have the asset.

If we consider this from the user's perspective, however, this lag may not be relevant or even known. Neither the receiving user nor the sending user may know when the asset arrived. If the lag between sending and receiving is a few seconds, it will be invisible or certainly tolerable to users who are directly communicating about the asset transfer. In this situation the system behavior is considered consistent and acceptable to the users, even though we are relying upon soft state and eventual consistency in the implementation.

EVENT-DRIVEN ARCHITECTURE

What if you do need to know when state has become consistent? You may have algorithms that need to be applied to the state but only when it has reached a consistent state relevant to an incoming request. The simple approach is to rely on events that are generated as state becomes consistent.

Continuing with the previous example, what if you need to notify the user that the asset has arrived? Creating an event within the transaction that commits the asset to the receiving user provides a mechanism for performing further processing once a known state has been reached. EDA (event-driven architecture) can provide dramatic improvements in scalability and architectural decoupling. Further discussion about the application of EDA is beyond the scope of this article.

CONCLUSION

Scaling systems to dramatic transaction rates requires a new way of thinking about managing resources. The traditional transactional models are problematic when loads need to be spread across a large number of components. Decoupling the operations and performing them in turn provides for improved availability and scale at the cost of consistency. BASE provides a model for thinking about this decoupling. Q

REFERENCES

1. http://highscalability.com/unorthodox-approach-database-design-coming-shard.
2. http://citeseer.ist.psu.edu/544596.html.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

DAN PRITCHETT is a Technical Fellow at eBay where he has been a member of the architecture team for the past four years. In this role, he interfaces with the strategy, business, product, and technology teams across eBay marketplaces, PayPal, and Skype. With more than 20 years of experience at technology companies such as Sun Microsystems, Hewlett-Packard, and Silicon Graphics, Pritchett has a depth of technical experience, ranging from network-level protocols and operating systems to systems design and software patterns. He has a B.S. in computer science from the University of Missouri, Rolla.