What belongs in state storage API’s?

- Problem: often can get by with less-than-ACID, but most widely-used state API’s (DB, fileys) don’t let you express constraints on consistency, ordering, etc.
  - Contrast Bayou: every update can carry a predicate and a conflict resolution function

- What about filesystem API?
  - Most apps deal with objects or data structures, not files
  - Filesystem API is impoverished (if you want to preserve interface compatibility)

- What about DB’s, which have “nicest” properties?
  - Often overkill, but cost of using is rarely exposed to developer (until system is deployed at scale...)

© 2002 Armando Fox
What kinds of API’s to think about?

- We should think in terms of object store/object access layer, not necessarily filesystem. DB is one kind of object store whose model is optimized to support a specific set of operations (relational queries).
  - What about providing support for RMW-type operations, rather than always separating “read” from “compute”?
  - UDF’s in Exokernel: “Here is a function that operates on…”
  - Could annotate function properties (deterministic? commutative? etc) and let storage subsystem schedule them
  - Challenge: desired ops likely to be app-specific; can we do something like this in an extensible manner? (recall Hellerstein’s GiST)
State storage API should reflect things about the implementation that would be really hard to add on top if they weren’t built in.

- What should be “expressible” by state storage API?
- Typically seen in customer apps (James): read-only, ACID, queued updates, near-real-time, time-travel, 2 phase commit
- Things that are hard to add after the fact: “Time travel” or versioning, atomic ops over groups of objects (vs. over single object)
- Challenge: should attributes be associated with data, or with methods?