Utilizing ACID atomicity/durability: queues

Queues:

- Transaction chopping
- Local queues
- Transactional queue patterns
- Locking mechanisms for queues
- Distributed queues

Queues in databases

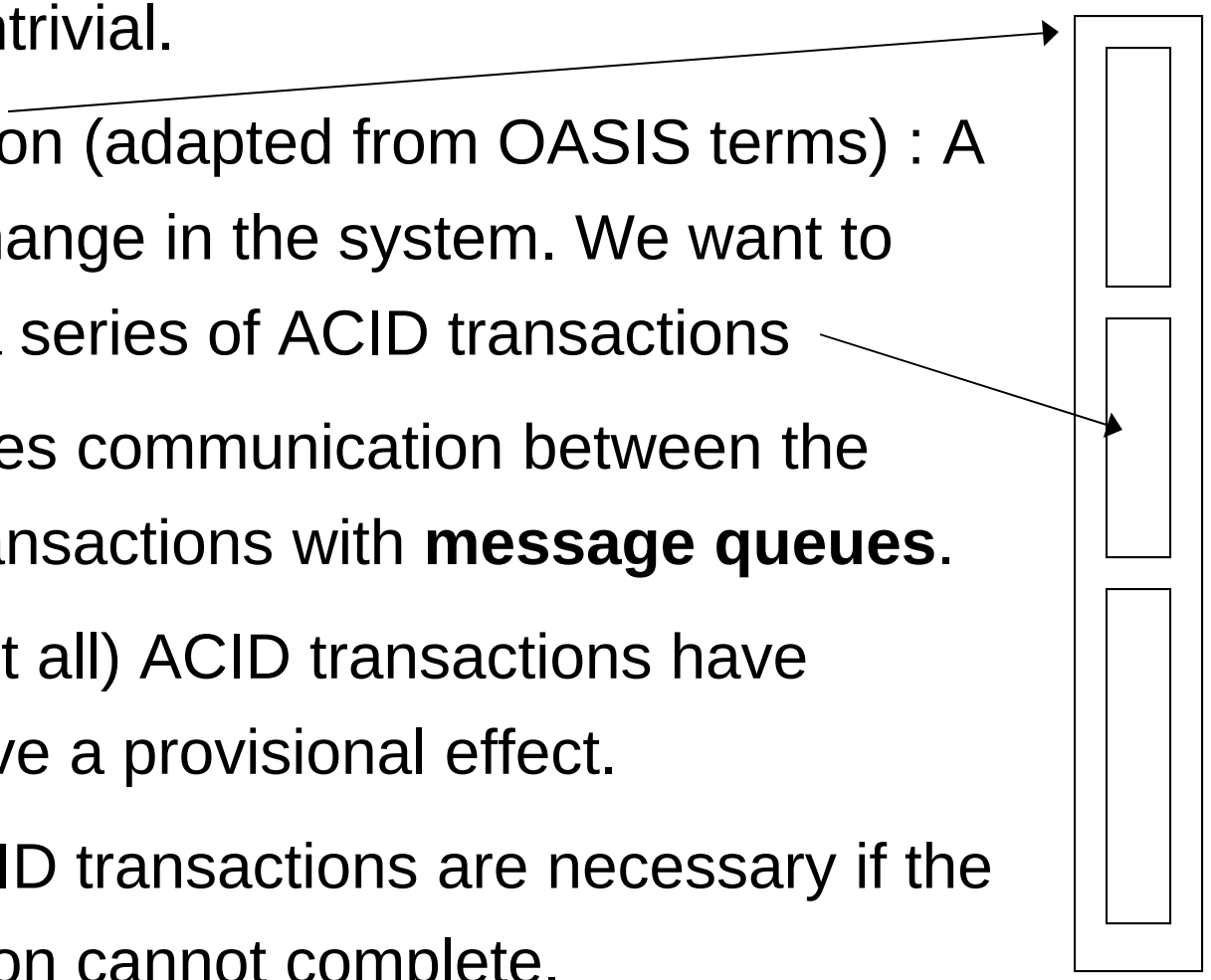
- A generic application of the ACID properties atomicity (and also durability.
- One technology that helps with many different problems:
  - Making transactions work in spite of aborts
  - Load buffering
  - Chopping complicated transactions
  - Secure messaging
    - The better Service Oriented Architecture (SOA)
  - Business process management
  - Stable user interaction with a transactional system.

Transaction chopping

- Is the technique of splitting one transaction into several ones. Consequences:
  - no ACID properties across the transaction boundary.
- Motivation: transactions should be only as long as necessary.

Chopping a business transaction

- Chopping one writing transaction into two writing transactions is nontrivial.
- Business transaction (adapted from OASIS terms) : A consistent state change in the system. We want to realize it through a series of ACID transactions
- In general it requires communication between the individual ACID transactions with **message queues**.
- After some (but not all) ACID transactions have terminated, we have a provisional effect.
- Compensating ACID transactions are necessary if the business transaction cannot complete.



Example of a business transaction

- In a transfer business transaction:
  - i.) withdraw $x from account y
  - ii.) put $x on account z
- Assume, a client program issues two transactions:
  - Transaction TA1: step i,
  - Transaction TA2: step ii,
- But what happens, if client program crashes after TA1?
- Solution:
  - TA1 leaves a note in a *message queue* and
  - Transactional queue access patterns

Example message queue

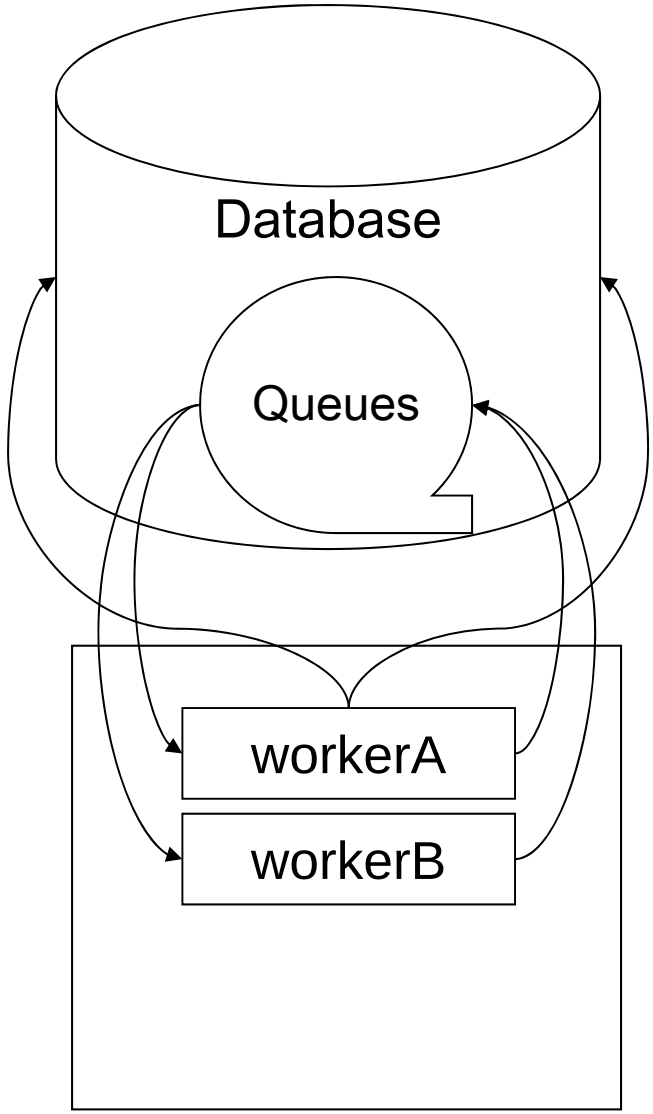
- In the transfer example:
  - Message queue is a table with scheme:
  - TransferOngoing(id, fromAccount, toAccount, amount)
  - TA1 leaves a row (id, y, z, x) in this queue
  - TA2 dequeues a row and does the appropriate action.
- Important: We will not and can not require strict first-in first-out (FIFO) processing of messages in a message queue.
- Processsing of messages in a best effort manner:
  - If possible, process older messages first.
  - Possible obstacles: aborts, concurrent access to message queue

Local message queues

- conceptual precursor to persistent message queues in distributed systems.
- A local message queue may be a table.
  - The rows are conceived as messages.
- Message producers: place messages in queue.
- Message consumers: they *dequeue* the messages, that means: remove messages from the queue.
- Dequeuing can be seen as an action on the level of the business logic: the message is not pending any more. The implementation could use at least two strategies: either deleting the message or marking it as processed (the consumer processes have to use the option chosen)

Architecture with queue

- The messages in the queues must be processed by database clients, the dequeue workers.
- These clients dequeue the message, process the message.
- They might optionally enqueue subsequent messages for further processing.



Messages as commands

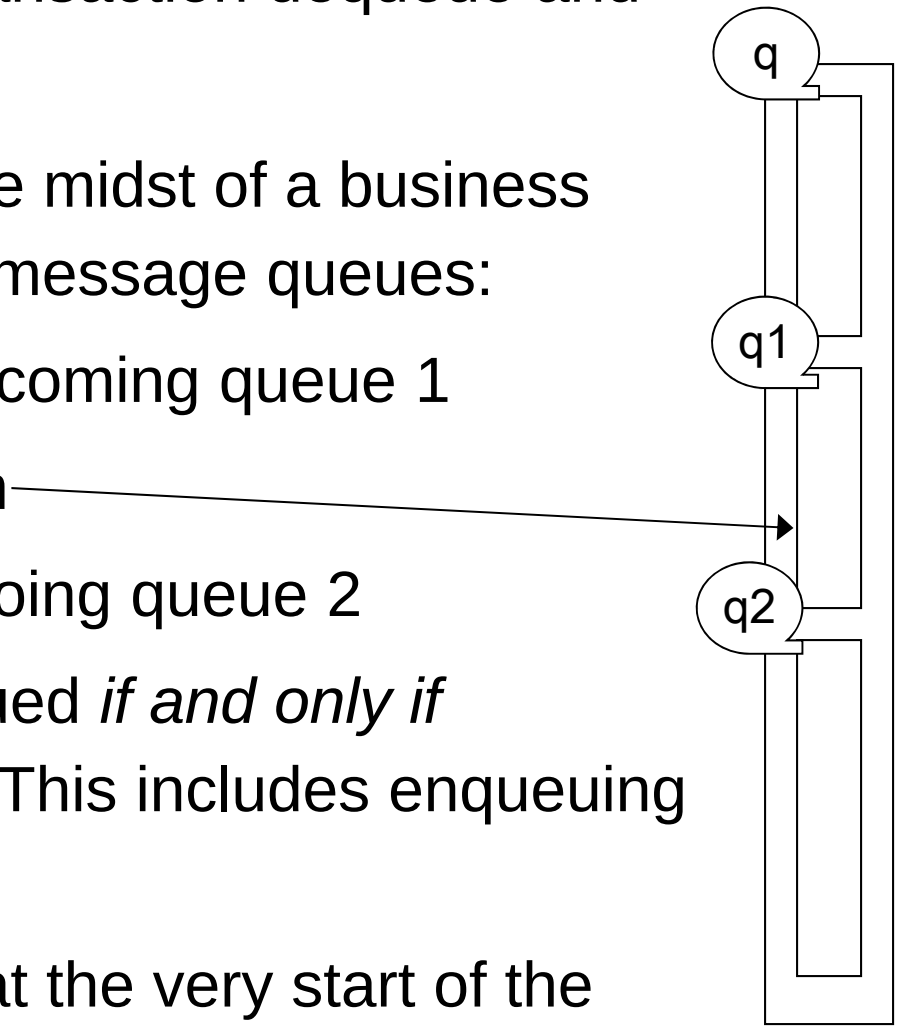
- Typical usage of message queues:
- Consumers (dequeue workers) are *activated* by messages:
  - A message demands action. If the action is taken, the message is dequeued.
- Messages are paired with appropriate actions, equivalent to pairing of method header and method body.
- In the simplest case one action is a subprogram that issues one ACID transaction to dequeue and process a message. We want to call this subprogram a *dequeue worker*.
- The dequeue worker gets activated by a different program, today often called container.

Transactional dequeue

- A common pattern of dequeueing access to a queue: *transactional dequeue pattern*:
- Performed by a dequeue worker, database client.
- A transactional dequeue for a local queue is a single local transaction that does two operations:
  - dequeue message from queue table
  - execute appropriate action
- Atomicity: Message is dequeued *if and only if* appropriate action succeeds.
- Transactional dequeue enables sophisticated transaction chopping, subsequent ACID transactions can communicate through the message queue in a safe manner.

Transaction chaining in business transactions

- Transactions in a business transaction dequeue and enqueue messages.
- A transactional dequeue in the midst of a business transaction may work on two message queues:
  - dequeue message from incoming queue 1
  - execute appropriate action
  - enqueue message in outgoing queue 2
- Atomicity: Message is dequeued *if and only if* appropriate action succeeds: This includes enqueuing of new messages.
- Typically there is one queue at the very start of the business transaction.



conditional response and declining

- In the example, TA1 might be withdrawal after check.
- The action is a conditional action, and has one outcome that is superficially equivalent to a rollback:
  - If insufficient funds are available, then the withdrawal is declined.
- Such an outcome is however a successful processing of the message:
  - The declining is the appropriate action.
  - However, in this application example the whole transfer must be declined. Again, this is the correct response.

Concurrent access to queues

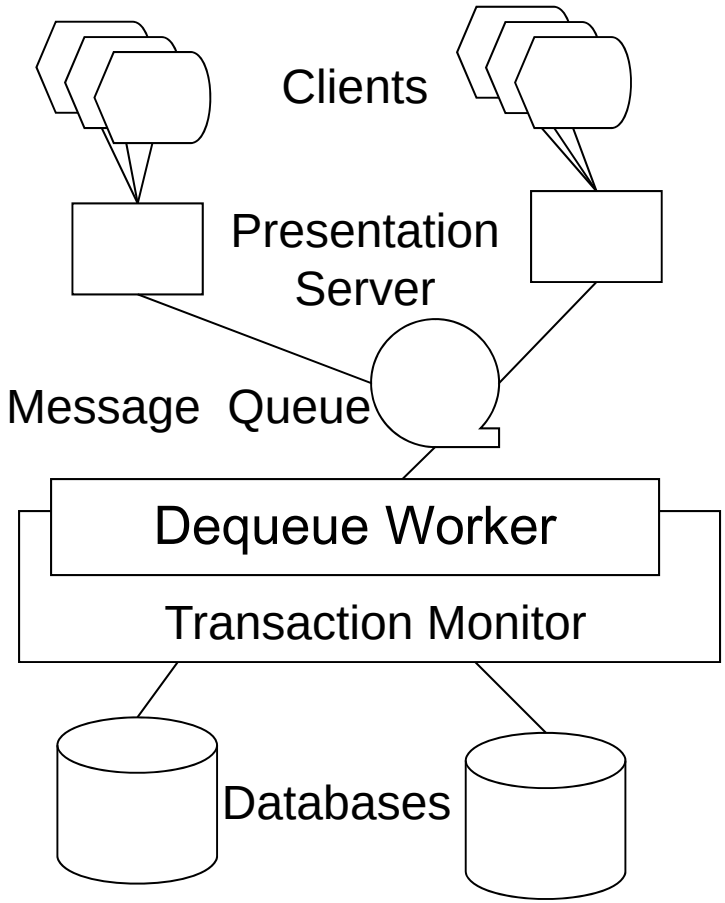
- Jim Gray 95:
- Queues are an *interesting database concept with interesting concurrency control.*
- Persistent queue systems need DBMS functionality.
- Queues are a powerful, interesting technology that motivate innovations in lock management.
- Desired operation: *read past*: go to the next unlocked item.
- Problem: It is tricky to look for the next unlocked object. If one looks at an object that is locked, usually the transaction gets blocked.

Simulating Read Past

- Queue Management and keeping track of unprocessed messages is done in a separate component, the dispatcher.
- Dispatcher uses isolation level “READ UNCOMMITTED”
- Dispatcher calls dequeue workers that do the transactional dequeue.
  - They receive the id of the message that they should work on as a parameter.
  - They only work on that message.
- This avoids the read-past problems.

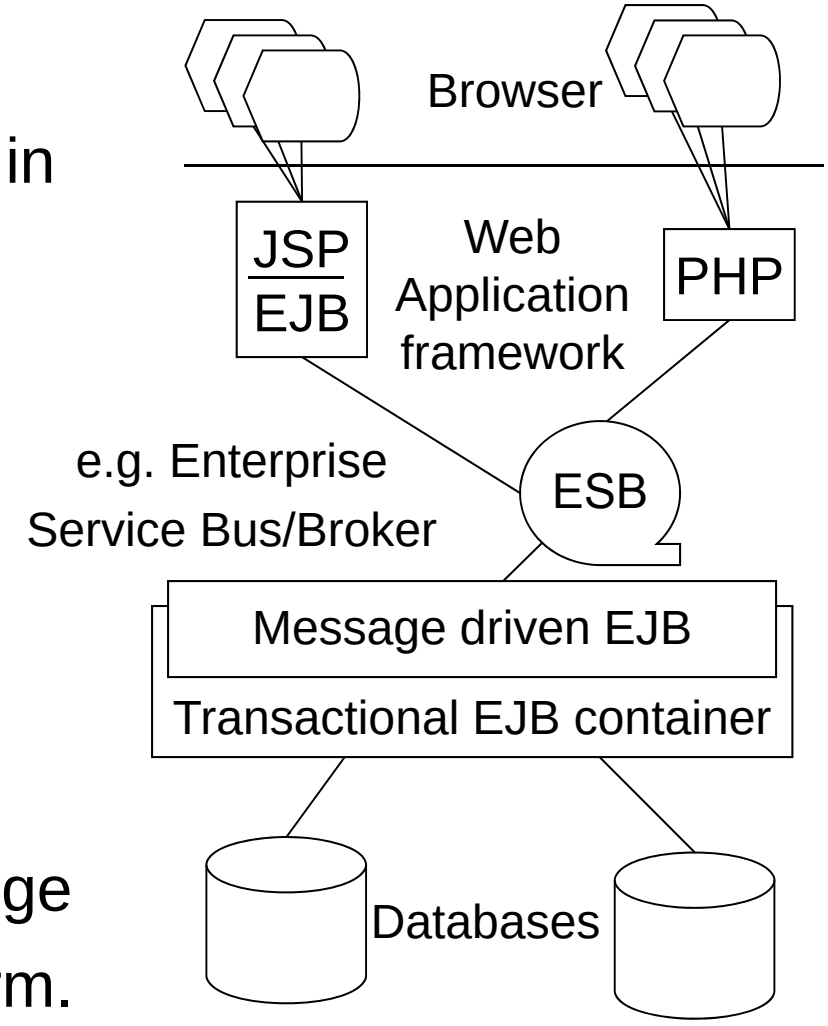
transactions, how it all began: flight reservation

- American Airlines and IBM started SABRE development 1960 – it is still a leading flight reservation system
- CICS: Classical computerized online transaction processing system
- 50,000 connected travel agencies
- Multi-tier architecture
- Today: Value of sold products: US$80 Billion



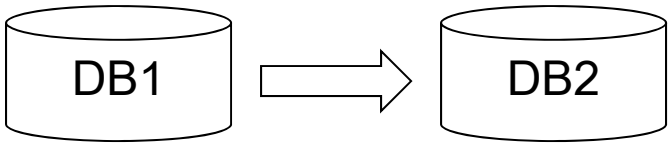
the old design prevails

- If used for fully developed TP heavy applications, modern enterprise platforms are used in the same way.
- Same design, different implementation.
- Todays enterprise application frameworks focus on component structure.
- But driving force may be change of underlying hardware platform.



Distributed messaging scenario

- Two databases, DB1 and DB2.
- Applications want to send messages from DB1 to DB2.
- Messages need to be processed on DB2.
- One queue on each DB:



  - on DB1: outbox1(messageID, status, message)
  - on DB2: inbox2(messageID, status, message)
- On outbox, only one worker W should work. It does not process the message, just moves it to inbox2.
- The inbox2 on DB2 has workers that process the message.
- W is a message consumer for outbox, and a message producer for inbox2.

Distributed messaging, secure delivery

- W has two independent database connections, one to DB1 and one to DB2. W does two ACID transactions.
- Only for outbox1 on DB1 it is a message consumer.
- W does a transactional dequeue of a message in DB1:

1. enqueues (writes) message in inbox2 on DB2.
2. commits this write on DB2.
3. commits the dequeue on DB1.

- If everything works, then the message is now in inbox2.
- What if W crashes after 2 and before 3 ?
- Dequeue on outbox is not committed.
- W will redo it, don’t we end up with multiple copies in DB2?

Idempotent operation of the worker

- An operation g is *idempotent*, if applying g twice has the same effect as applying g once.
- The enqueue of the message to inbox2 (step 1 and 2) should be an idempotent enqueue attempt:
  - If a message with the same message id is already in inbox2, skip the enqueue.

- This procedure is possible, because the operation of providing a piece of information is inherently idempotent.
- E.g. incrementing a counter would not be idempotent.

Idempotence is crucial point of this protocol

- Because of the idempotence of the enqueue, it is possible to do reach distributed ACID properties with two ACID transactions:
- The two databases do not need to know that they are part of a distributed, transactional communication.
- General distributed ACID transactions are MUCH more complicated and heavyweight.
  - Require a special voting protocol (Two-phase commit, not to be confused with 2Phase locking)
  - Require special infrastructure, are risky.
- Distributed messaging are much more lightweight

Connection to Service Oriented Architecture

- Service Oriented Architecture(SOA): A system architecture where the system is built from components communicating over service interfaces (web services are just one example).
- This makes the components reusable, since the service interfaces can be connected in new topographies.
- Service interfaces often have a messaging flavor.
- Important requirement for service interfaces: idempotence.
- If a message is sent twice, it should have the same effect as if it is sent only once.
- Often equivalent to using an id: Two messages with the same id should be identical, are treated as single message.

Message queues as load buffer

- Message queues are placed at the boundary of a high-performance computing zone.
- Outside world (User) places requests to the transaction service into the message queue:
  - Once queued, the message will be processed.
  - Reliability for the user.
- Purpose of the application server: continuously processes pending requests (= messages) in the queue.
- In high load times, the application server is 100% utilized.
- Several application servers can work on the same queue.