Modern Database Systems

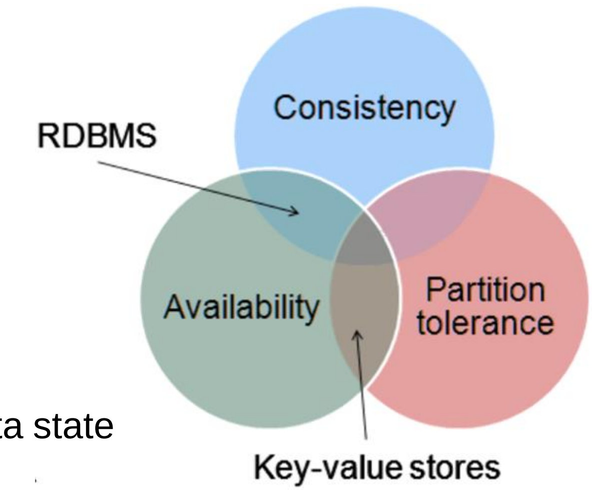
Advanced Aspects of Big Data Management

Doc. RNDr. Irena Holubova, Ph.D.
Irena.Holubova@matfyz.cuni.cz

CAP Theorem

CAP Theorem

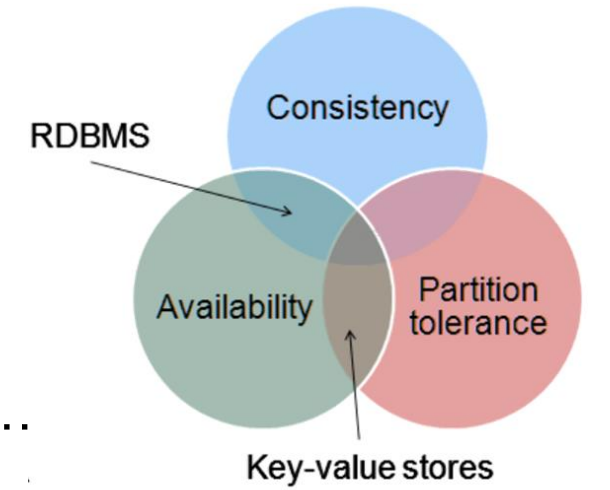
Recapitulation



- **Consistency**
  - Consistent reads and writes
  - Concurrent operations see the same valid and consistent data state
- **Availability**
  - The system is available to serve at the time when it is needed
  - Node failures do not prevent survivors from continuing to operate
- **Partition tolerance**
  - The ability of a system to continue to service in the event a few of its cluster members become unavailable
- **Theorem:** In systems that are distributed or scaled out it is impossible to achieve all three.
  - First appeared in 1998, published in 1999
  - Established as theorem and proved in:

    Lynch, Nancy and Gilbert, Seth. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. ACM SIGACT News, volume 33 issue 2, 2002, pages 51-59.

CAP Theorem
Recapitulation



- **Consistency + Availability**
  - Single-site databases, cluster databases, …
- **Consistency + Partition Tolerance**
  - Distributed databases, distributed locking, majority protocols, …
- **Availability + Partition Tolerance**
  - Web caching, DNS

- Examples:
  - RDBMS: CA
  - Apache Cassandra (column): AP
  - Google BigTable (column): CA
  - Apache CouchDB (document): AP

CAP Theorem

Proof

- Formalization of the notion of consistency, availability and partition tolerance:
  - Atomic Consistency, Atomic Data Object
    - There must exist a total order on all operations such that each operation looks as if it were completed at a single instant
      - i.e., any read operation that begins after a write operation completes must return that value
      - Equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time
  - Available Data Object
    - Every request received by a non-failing node in the system must result in a response
      - i.e., any algorithm must eventually terminate
      - Although we do not say how long it will take
  - Partition Tolerance
    - The network is allowed to lose arbitrarily many messages sent from one node to another

CAP Theorem

Proof

No clock, nodes make decisions on the basis of **messages** and **local computations**

Theorem 1. It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:

- Availability
- Atomic consistency

in all fair executions (including those in which messages are lost).

Proof (by contradiction). Assume an algorithm A exists that meets the three criteria: atomicity, availability, and partition tolerance. We construct an execution of A in which there exists a request that returns an inconsistent response.

- Assume that the network consists of at least two nodes $\Rightarrow$ it can be divided into two disjoint, non-empty sets G_1 and G_2.
- Assume that all messages between G_1 and G_2 are lost.
- If a write occurs in G_1, and later a read occurs in G_2, then the read operation cannot return the results of the earlier write operation.

More formally…

CAP Theorem

Proof

- Let v_0 be the initial value of the atomic object.
- Let α_1 be the prefix of an execution of A in which a single write of a value not equal to v_0 occurs in G_1, ending with the termination of the write operation.
- Assume that no other client requests occur in either G_1 or G_2, no messages from G_1 are received in G_2 and vice versa.
- We know that this write completes, by the availability requirement.
- Let α_2 be the prefix of an execution in which a single read occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- During α_2 no messages from G_2 are received in G_1 and vice versa.
- Again we know that the read returns a value by the availability requirement. The value returned by this execution must be v_0, as no write operation has occurred in α_2.
- Let α be an execution beginning with α_1 and continuing with α_2.
- To the nodes in G_2, α is indistinguishable from α_2, as all the messages from G_1 to G_2 are lost, and α_1 does not include any client requests to nodes in G_2. Therefore in the α execution, the read request (from α_2) must still return v_0.
- However the read request does not begin until after the write request (from α_1) has completed. This therefore contradicts the atomicity property, proving that no such algorithm exists.

■

CAP Theorem

Proof

Corollary. It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:

- Availability, in all fair executions,
- Atomic consistency, in fair executions in which no messages are lost.

Proof. Main idea: In the asynchronous model an algorithm has no way of determining whether a message has been lost, or has been arbitrarily delayed (atomic consistency when no messages are lost => atomic consistency in all executions => violation of Theorem 1).

- For the sake of contradiction assume that there exists an algorithm A that always terminates, and guarantees atomic consistency in fair executions in which all messages are delivered.
- Theorem 1 implies that A does not guarantee atomic consistency in all fair executions, so there exists some fair execution of A in which some response is not consistent. I.e., at some finite point α in execution the algorithm A returns a response that is not atomic consistent.
- Let α' be the prefix of ending with the invalid response. Next, extend α' to a fair execution α'', in which all messages are delivered. The execution α'' is now a fair execution in which all messages are delivered. However this execution is not atomic. Therefore no such algorithm A exists.

■

CAP Theorem

Proof

- In the real world, most networks are not purely asynchronous
- Partially Synchronous Networks
  - Each node in the network has a clock
  - All clocks increase at the same rate
  - The clocks are not synchronized

$\Rightarrow$ Clocks = timers = can measure how much time has passed

  - Can be used for scheduling
  - Every message is either delivered within a given, known time t_{msg} or it is lost

CAP Theorem

Proof

Theorem 2. It is impossible in the partially synchronous network model to implement a read/write data object that guarantees the following properties:

- Availability
- Atomic consistency

in all executions (even those in which messages are lost).

Proof (similar to Theorem 1). We divide the network into two components G_1 and G_2, and construct an admissible execution in which a write happens in one component, followed by a read operation in the other component. This read operation can be shown to return inconsistent data.

- More formally…

CAP Theorem

Proof

- We will construct execution α_1 as in Theorem 1: a single write request and acknowledgment in G_1, whereas all messages between G_1 and G_2 are lost.
- We will construct the second execution α'_2 slightly differently: Let α'_2 be an execution that begins with a long interval of time during which no client requests occur. This interval must be at least as long as the entire duration of α_1.
- Then append to α'_2 the events of α_2, as defined in Theorem 1: a single read request and response in G_2, again assuming all messages between the two components are lost.
- Finally, construct by superimposing the two executions α_1 and α'_2.
- The long interval of time in α_2 ensures that the write request in α_1 completes before the read request in α'_2 begins.
- However, as in Theorem 1, the read request returns the initial value, rather than the new value written by the write request, violating atomic consistency.

■

CAP Theorem

Proof

- Analogue of corollary does not hold in partially synchronous model
  - The proof depends on nodes being unaware of when a message is lost

Managing Transactions

Managing Transactions

- Critics of NoSQL databases focus on the lack of support for transactions
- Business transaction
  - e.g., browsing a product catalogue, choosing a bottle of Talisker at a good price, filling in credit card information, and confirming the order
- System transaction
  - At the end of the interaction with the user
  - Locks are only held for a short period of time
- Business transaction = a series of system transactions

Managing Transactions

- Offline concurrency involves manipulating data for a business transaction that spans multiple data requests
  - Having a system transaction open for the whole business transaction is not usually possible
    - Long system transactions are not supported
- Problems:
  - Overwriting uncommitted data
    - More transactions select the same row and then update the row based on the value originally selected unaware of the other
  - Reading uncommitted data
    - A transaction accesses the same row several times and reads different data each time
- i.e., calculations and decisions may be made based on data that is changed
  - e.g., price list may be updated, someone may update the customer’s address, changing the shipping charges, …

$$S = \begin{array}{c} \begin{array}{cc} T1 & T2 \end{array} \\ \begin{bmatrix} W(A) & \\ & W(B) \\ W(B) & \\ Com. & \\ & W(A) \\ & Com. \end{bmatrix} \end{array}$$

overwriting uncommitted data
(blind write)

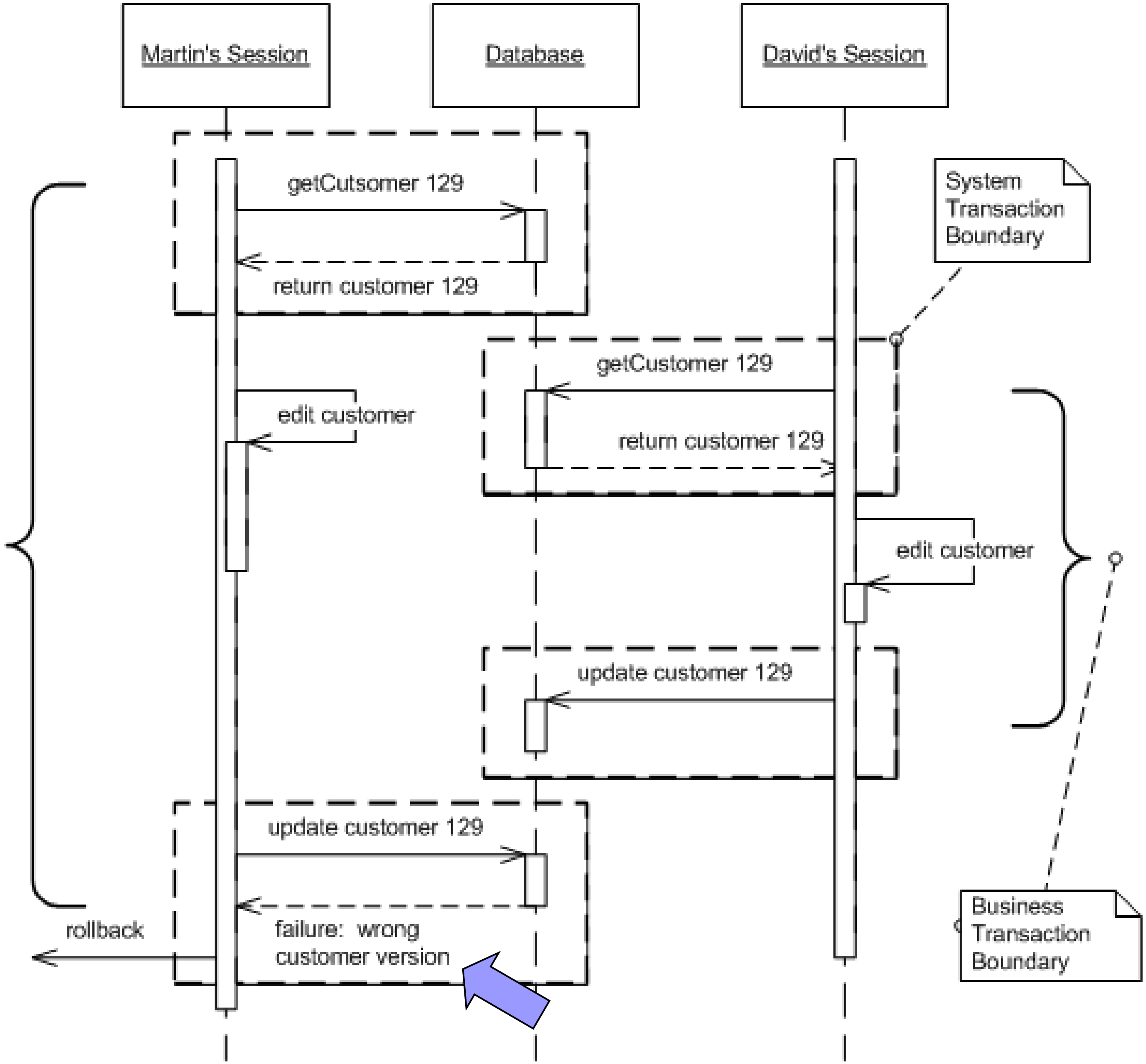
$$S = \begin{array}{c} \begin{array}{cc} T1 & T2 \end{array} \\ \begin{bmatrix} R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ & R(B) \\ & W(B) \\ & Com. \\ R(B) & \\ W(B) & \\ Com. & \end{bmatrix} \end{array}$$

reading uncommitted data
(dirty read)

Managing Transactions

Optimistic Offline Lock

- Assumes that the chance of conflict is low
- A form of conditional update
  - Ensures that changes about to be committed by one session do not conflict with the changes of another session
- Pre-commit validation
  1. Client operation re-reads any information that the business transaction relies on
  2. It checks that it has not changed since it was originally read and displayed to the user
- Obtaining a lock indicating that it is okay to go ahead with the changes to the record data

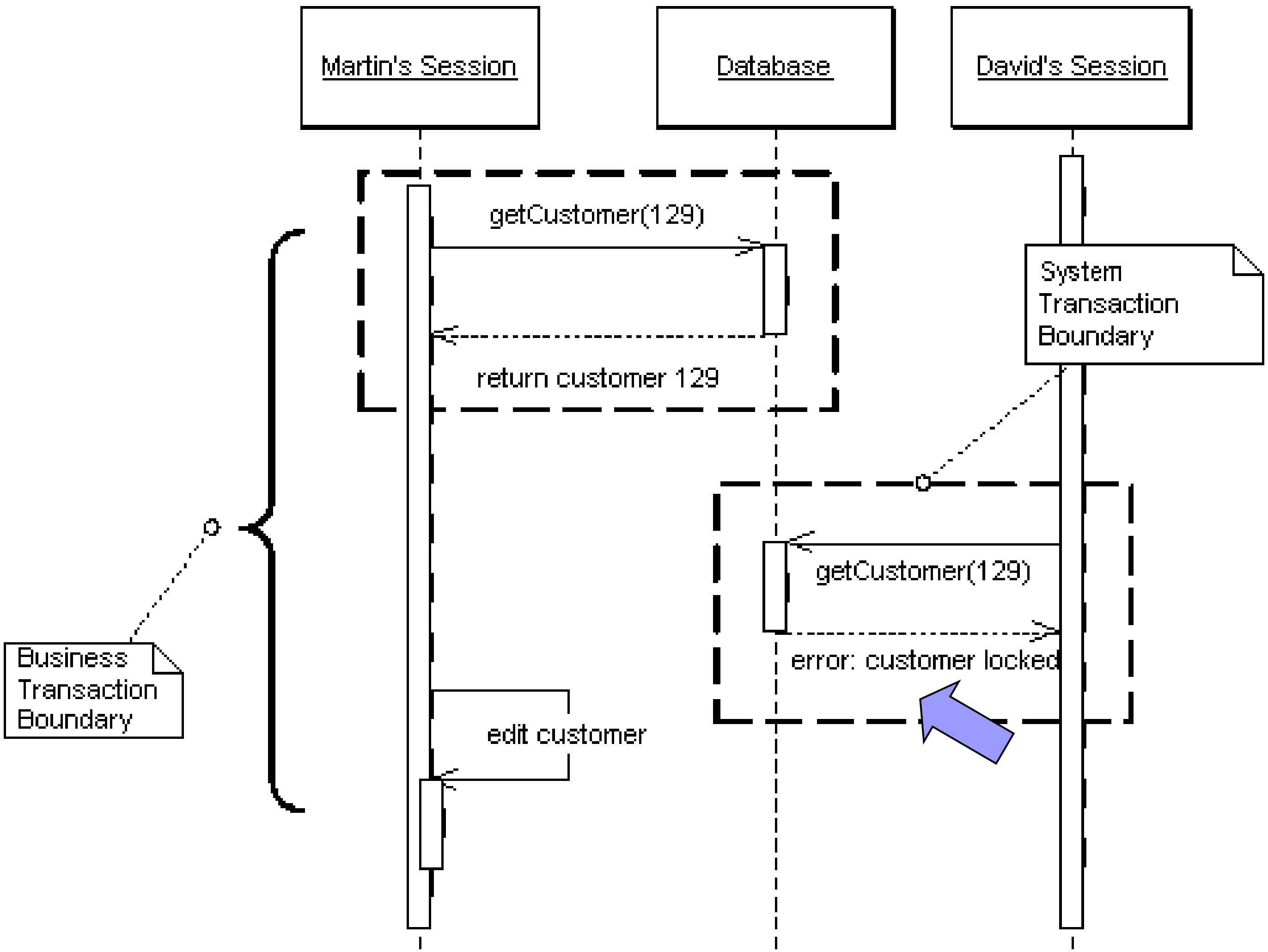


Martin's Session
Database
David's Session
getCutsomer 129
return customer 129
System Transaction Boundary
getCustomer 129
edit customer
return customer 129
edit customer
update customer 129
update customer 129
rollback
failure: wrong customer version
Business Transaction Boundary

Managing Transactions

Pessimistic Offline Lock

- Problems of optimistic approach:
  - There might be many conflicts
  - The conflict can be detected at the end of a lengthy business transaction
- Pessimistic solution: allows <u>only one business transaction</u> at a time to <u>access data</u>
- Forces a business transaction to acquire a lock on each piece of data before it starts to use it
  - Once a business transaction begins, it surely completes
- Lock manager
  - Simple, single (for all business transactions), centralized (or based on the database in the distributed system)
- Standard issue: deadlock
  - Timeout for an application
    - Automatically rolled-back after a period of time of non responding
  - Timestamp attribute for a lock
    - Automatically released after a period of time

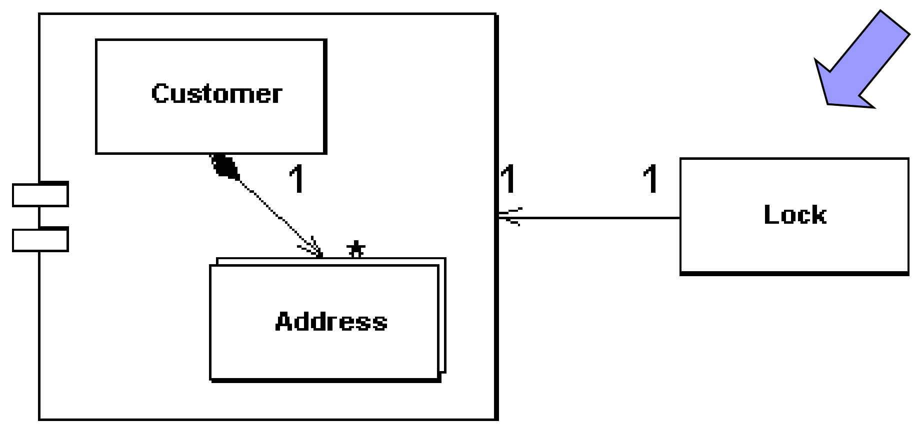


Martin's Session
Database
David's Session
getCustomer(129)
return customer 129
System Transaction Boundary
getCustomer(129)
error: customer locked
Business Transaction Boundary
edit customer

Managing Transactions

Coarse-grained Lock

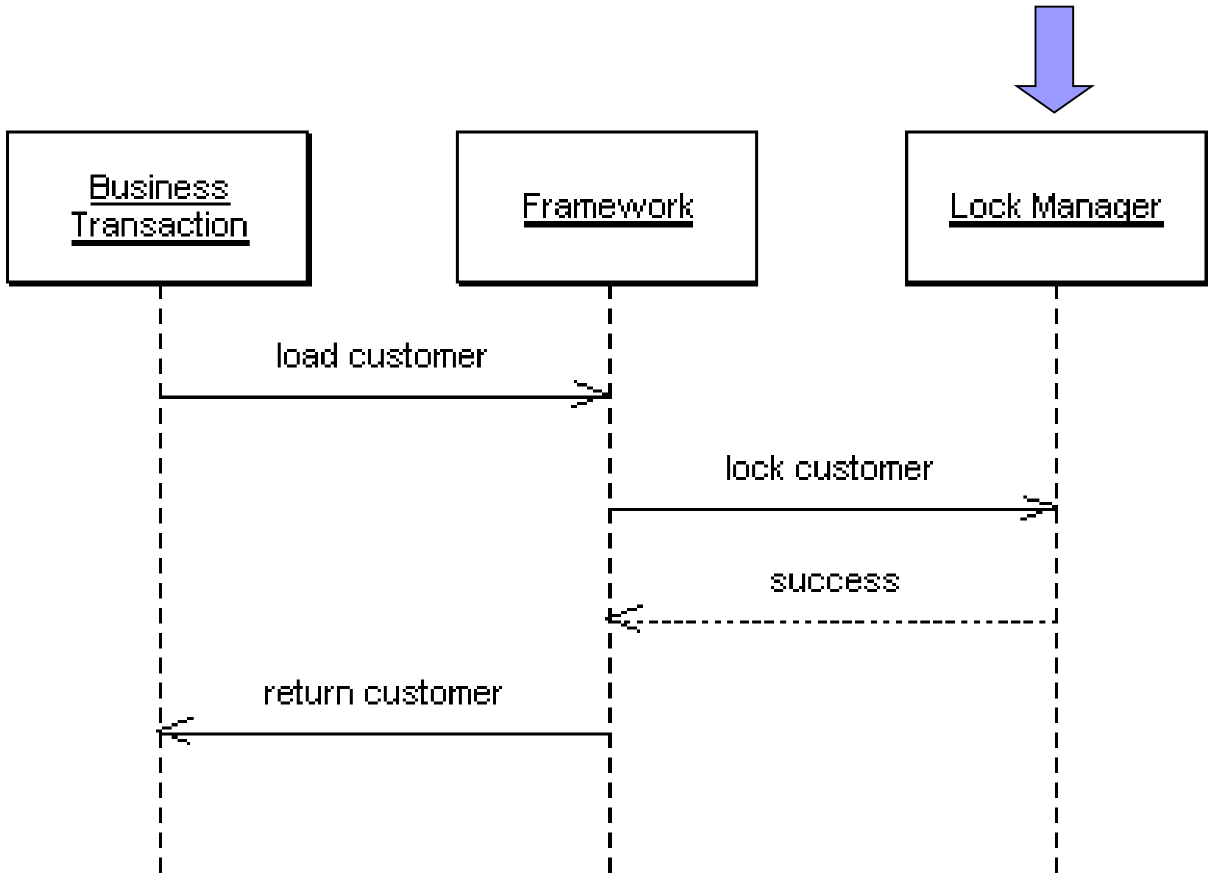
- When objects are edited as a group
  - Logically related objects
  - e.g., a customer and its set of addresses
    - We want to lock any one of them
- A separate lock for individual objects presents a number of challenges
  - We need to find them all in order to lock them
    - Gets tricky as we get more locking groups
    - When the groups get complicated
      - Nested groups
- Idea: a single lock that covers many objects
  - A sophisticated lock manager



Managing Transactions

Implicit Lock

- Problem: forgetting to write a single line of code that acquires a lock $\Rightarrow$ entire offline locking scheme is useless
  - Failing to retrieve a read lock $\Rightarrow$ other transactions use write locks $\Rightarrow$ not getting up-to-date session data
  - Failing to use a version count $\Rightarrow$ unknowingly writing over someone's changes
  - Not releasing locks $\Rightarrow$ bring productivity to a halt
- Fact: If an item might be locked anywhere it must be locked everywhere
- Idea: <u>locks are automatically acquired</u>
  - Not explicitly by developers but implicitly by the application


Business Transaction
Framework
Lock Manager
load customer
lock customer
success
return customer

Performance Tuning

Performance Tuning

Goals

Example from 2010: Tweets add up to 12 Terabytes per day. This amount of data needs around 48 hours to be written to a disk at a speed of about 80 Mbps.

- MapReduce creates a bottleneck-free way of scaling out
- To reduce latency
  - Latency:
    - Non-parallel systems: time taken to execute the entire program
    - Parallel systems: time taken to execute the smallest atomic sub-task
  - Strategies:
    - Reducing the <u>execution time</u> of a program
    - Choosing the most <u>optimal algorithms</u> for producing the output
    - <u>Parallelizing</u> the execution of sub-tasks
- To increase throughput
  - Throughput = the amount of input that can be manipulated to generate output within a process
  - Non-parallel systems:
    - Constrained by the available resources (amount of RAM, number of CPUs)
  - Parallel systems:
    - “No” constraints
    - Parallelization allows for any amount of commodity hardware

Performance Tuning

Linear Scalability

- Typical horizontally scaled MapReduce-based model: linear scalability
  - “One node of a cluster can process ***x*** MBs of data every second $\rightarrow$ ***n*** nodes can process ***x*** × ***n*** amounts of data every second.”
    - Time taken to process *y* amounts of data on a single node = t seconds
    - Time taken to process *y* amounts of data on *n* nodes = *t* / *n* seconds
- Assumption: tasks can be parallelized into equally balanced units

Performance Tuning

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

Amdahl's Law

- Formula for <u>finding the maximum improvement</u> in performance of a system when a part is improved
  - *P* = the proportion of the program that is parallelized
  - 1 – *P* = the proportion of the program that cannot be parallelized
  - *N* = the times the parallelized part performs as compared to the non-parallelized one
    - i.e., how many times faster it is
      - e.g., the number of processors
    - Tends to infinity in the limit
- Example: a process that runs for 5 hours (300 minutes); all but a small part of the program that takes 25 minutes to run can be parallelized
  - Percentage of the overall program that can be parallelized: 91.6%
  - Percentage that cannot be parallelized: 8.4%
  - Maximum increase in speed: 1 / (1 – 0.916) = ~11.9 times faster
    - *N* tends to infinity

Performance Tuning

$L = kW$

Little's Law

- Origins in economics and queuing theory (mathematics)
- Analyzing the load on stable systems
  - Customer joins the queue and is served (in a finite time)
- "The average number of customers (L) in a stable system is the product of the average arrival rate (k) and the time each customer spends in the system (W)."
  - Intuitive but remarkable result
  - i.e., the relationship is not influenced by the arrival process distribution, the service distribution, the service order, or practically anything else
- Example: a gas station with cash-only payments over a single counter
  - 4 customers arrive every hour
  - Each customer spends about 15 minutes (0.25 hours) at the gas station

  $\Rightarrow$ There should be on average 1 customer at any point in time

  $\Rightarrow$ If more than 4 customers arrive at the same station, it would lead to a bottleneck

Performance Tuning

Message Cost Model

initialization

$C = a + bN$

linear dependence on size

- Breaks down <u>the cost of sending a message</u> from one end to the other in terms of its fixed and variable costs
  - *C* = cost of sending the message from one end to the other
  - *a* = the upfront cost for sending the message
  - *b* = the cost per byte of the message
  - *N* = number of bytes of the message
- Example: gigabit Ethernet
  - *a* is about 300 microseconds = 0.3 milliseconds
  - *b* is 1 second per 125 MB
    - Implies a transmission rate of 125 MBps.
  - 100 messages of 10 KB => take 100 × (0.3 + 10/125) ms = 38 ms
  - 10 messages of 100 KB => take 10 × (0.3 + 100/125) ms = 11 ms
  - A way to optimize message cost is to send as big packet as possible each time

0,08

0,8

References

- Pramod J. Sadalage - Martin Fowler: **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**
- Eric Redmond - Jim R. Wilson: **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**
- Sherif Sakr - Eric Pardede: **Graph Data Management: Techniques and Applications**
- Shashank Tiwari: **Professional NoSQL**
- **http://martinfowler.com/**