LARGE SCALE DATA STORAGE

Gergely Magyar, PhD.
Center for Intelligent Technologies
Department of Cybernetics and Artificial Intelligence
Technical University of Košice

DCAI
Department of Cybernetics and Artificial Intelligence

CIT
Center for Intelligent Technologies

CONTENTS

- MapReduce
- CAP theorem and eventual consistency
- distributed key-value store
- scalable databases
- Microsoft Azure HDInsight

MAPREDUCE - MOTIVATION

- many tasks composed of processing lots of data to produce lots of other data
- large scale data processing
  - want to use 1000s of CPUs
  - (but don’t want hassle of managing things)

LARGE SCALE DATA PROCESSING

- MapReduce provides:
  - user-defined functions
  - automatic parallelization and distribution
  - fault tolerance
  - I/O scheduling
  - status and monitoring

CHALLENGES WITH TRADITIONAL PROGRAMMING MODELS (MPI)

- MPI gives you MPI_send, MPI_receive
- deadlock is possible …
  - Proc1: MPI_receive(Proc2, A), MPI_send(Proc2, B)
  - Proc2: MPI_receive(Proc1, B), MPI_send(Proc1, A)
- large overhead from communication mismanagement
  - time spent blocking is wasted cycles
  - can overlap computation with non-blocking communication
- load imbalance is possible! Dead machines?
- things are starting to look hard to code!

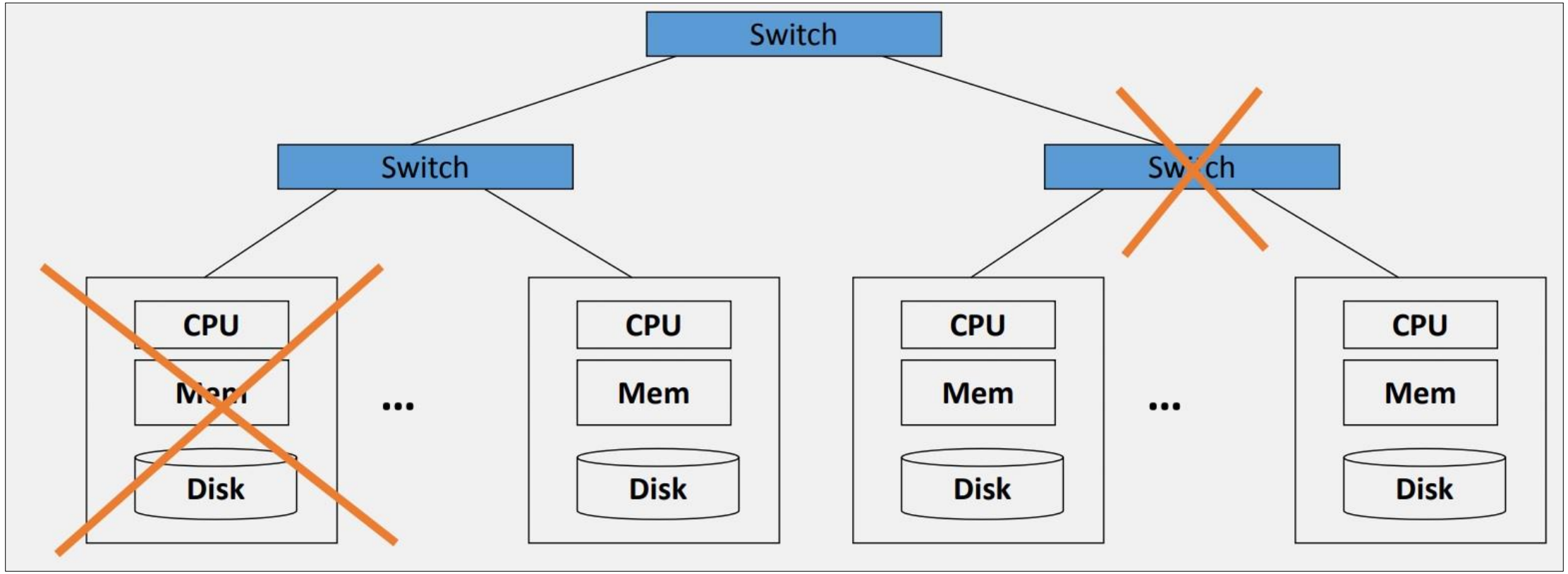
COMMODITY CLUSTERS

- web data sets can be very large
  - tens to hundreds of terabytes
  - cannot mine on a single server
- standard architecture emerging:
  - cluster of commodity Linux nodes
  - Gigabit Ethernet interconnect
- how to organize computations on this architecture?
  - mask issues such as hardware failure

SOLUTION

- use distributed storage
  - 6-24 disks attached to a blade
  - 32-64 blades in a rack connected by Ethernet
- push computations down to storage
  - computations process contents of disks
  - data on disks read sequentially from beginning to end
  - rate limited by speed of disks

CLUSTER ARCHITECTURE



STABLE STORAGE

- first-order problem: if nodes can fail, how can we store data persistently?
- answer: distributed file system
  - provides global file namespace
  - Google GFS / Hadoop HDFS
- typical usage pattern
  - huge files (100s of GB to TB)
  - data is rarely updated in place
  - reads and appends are common

WHAT IS MAPREDUCE

- MapReduce
  - programming model from LISP
- Many problems can be phrased this way
- **easy** to distribute
  - hides difficulty of writing parallel code
  - system takes care of load balancing, dead machines, etc.
- nice retry and failure semantics

PROGRAMMING CONCEPT

- Map
  - **perform** a function on **individual values** in a data set to create a **new list** of values
- Reduce
  - **combine** values in a data set to create a new **value**

SQUARE X = X * X **MAP** SQUARE [1,2,3,4,5] RETURNS [1,4,8,16,25]	SUM= [All Elements in Array, Total+=] REDUCE [1,2,3,4,5] RETURNS 15

MAPREDUCE PROGRAMMING MODEL

- **input and output – each a set of key/value pairs**
- **programmer specifies two functions:**
- MAP (IN_KEY, IN_VALUE)
  LIST(OUT_KEY, INTERMEDIATE_VALUE)
- processes input key/value pair
- produces intermediate pairs

MAPREDUCE PROGRAMMING MODEL

- **input and output – each a set of key/value pairs**
- **programmer specifies two functions:**
- REDUCE (IN_KEY, IN_VALUE)
  LIST(OUT_KEY, INTERMEDIATE_VALUE)
- combines all intermediate values for a particular key
- produces a set of merged output values (usually just one)

WORD COUNT EXAMPLE

- have a large file of words, many words in each line
- count the number of times each distinct word appears in the file

WORD COUNT PROGRAM

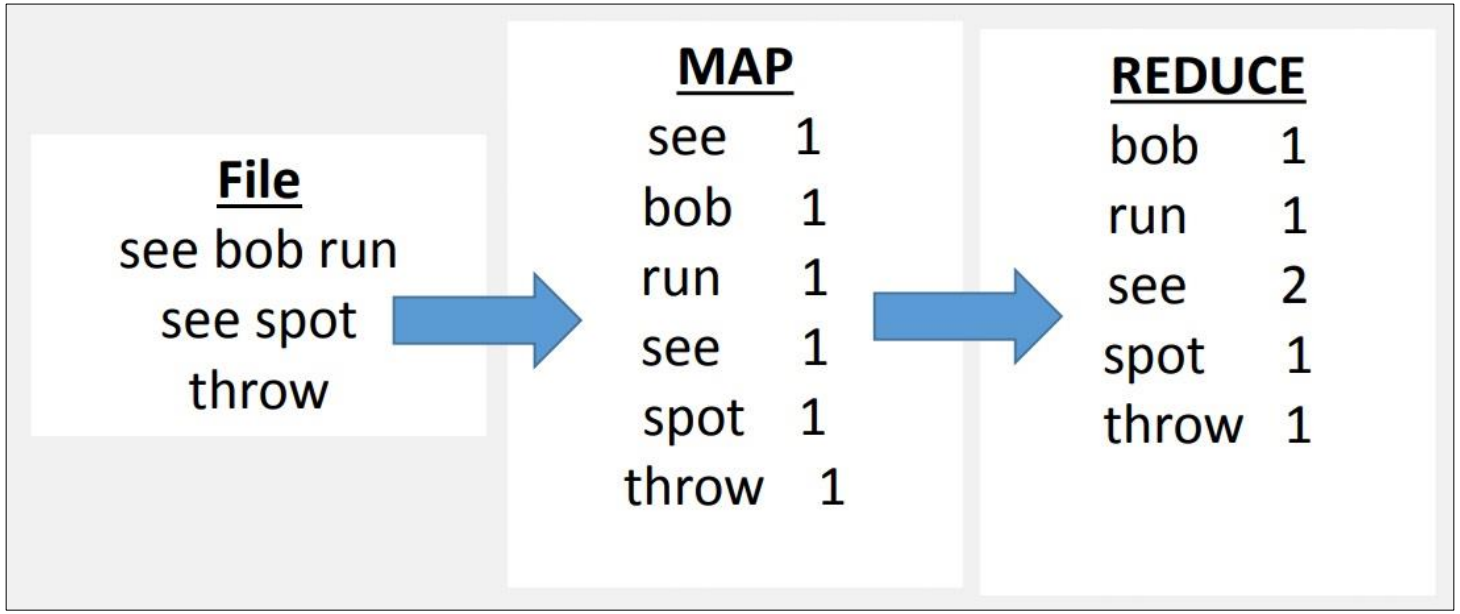
```
MAP(KEY = LINE, VALUE = CONTENTS):
        FOR EACH WORD W IN VALUE:
                EMIT INTERMEDIATE(w, 1)

REDUCE(KEY, VALUES):
//key: a word; values: an iterator
//over counts
        RESULT = 0
        FOR EACH V IN INTERMEDIATE VALUES:
                RESULT += V
        EMIT(KEY, RESULT)
```

WORD COUNT ILLUSTRATED

```
map(key=line, values=contents):
   for each word w in contents:
    emit(w,"1")
```

```
reduce(key=line, values=uniq_counts):
   sum all 1's in values list
     emit result (word, sum)
```



MAPREDUCE PROS AND CONS

- now it's easy to program for many CPUs
  - communication management effectively gone
  - fault tolerance, monitoring
  - can be much easier to design and program
  - can cascade several MapReduce tasks
- but … it further restricts solvable problems
  - might be hard to express a problem in MapReduce
  - Data parallelism is key

MAPREDUCE CONCLUSIONS

- MapReduce has proven to be a useful abstraction
- greatly simplifies large-scale computations
- functional programming paradigm can be applied to large-scale applications
- fun to use: focus on problem, let the middleware deal with messy details

CONSISTENCY IN A DISTRIBUTED SYSTEM

- why is consistency such a big deal in a distributed system
- why can reaching consistency be slow in a distributed system
- what components in a cloud are impacted by consistency issues
- solutions – consistency models
- how are components in a cloud built using these solutions
- practical issues

FALLACIES OF DISTRIBUTED COMPUTING (PETER DEUTSCH)

- the network is reliable
- latency is zero
- bandwidth is infinite
- the network is secure
- topology doesn’t change
- there is one administrator
- transport cost is zero
- the network is homogeneous

EFFECTS OF FALLACIES

- application and transport-layer developers allow unbounded traffic, greatly increasing dropped packets and wasting bandwidth through network latency and packet loss
- network security complacency allows malicious users and programs that adapt to security measures
- multiple administrators, as with subnets for rival companies, may use conflicting policies
- building and maintaining a network/subnet incurs large “hidden” costs
- bandwidth limits on the part of traffic senders can result in bottlenecks over frequency-multiplexed media

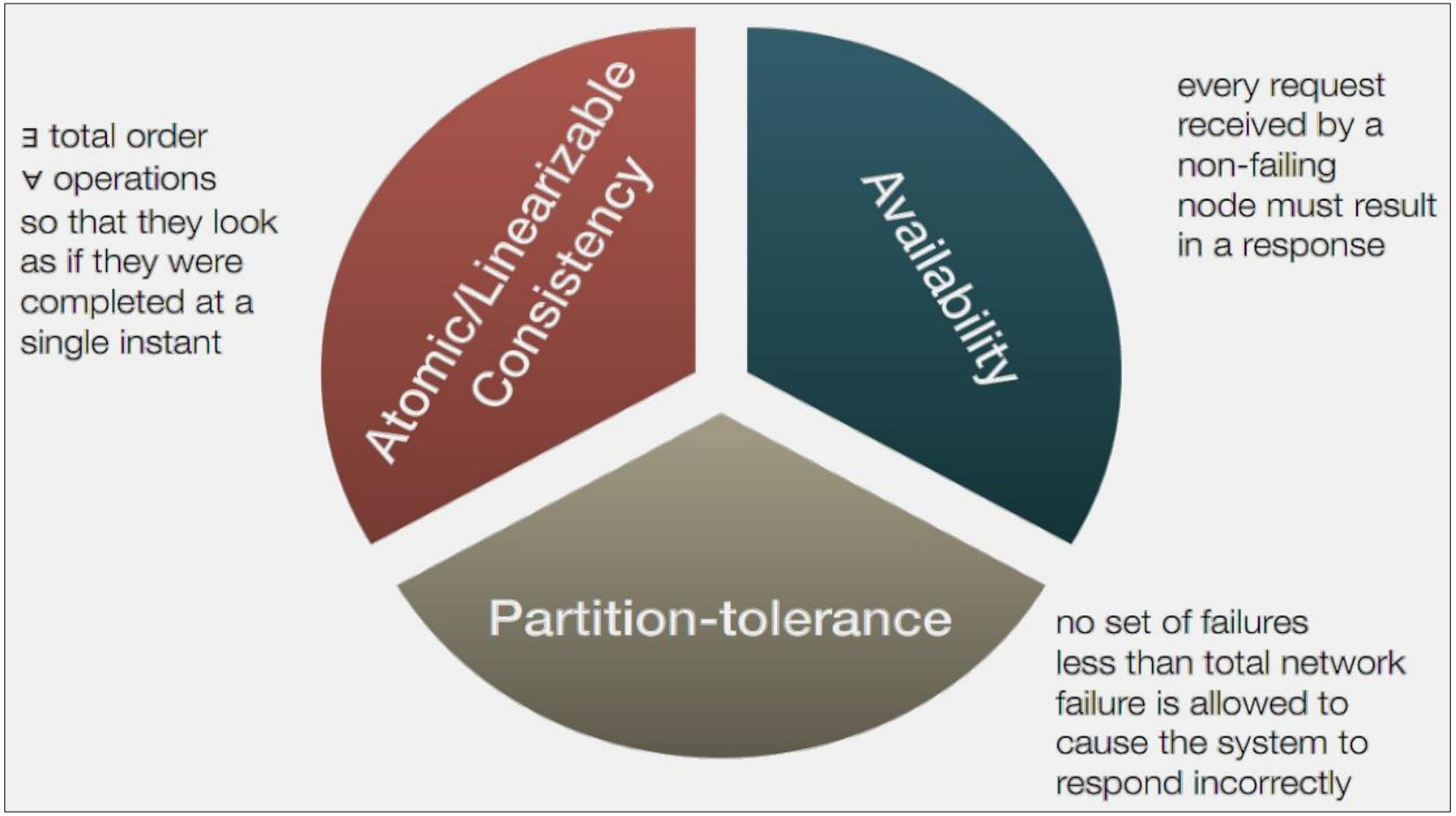
ERIC BREWER'S CAP THEOREM

- "you can have just two of the *consistency, availability* and *partition tolerance*"
  - data centers need to be very responsive, hence availability is vital
  - responsiveness – even if a transient fault makes it hard to reach some service
  - use cached data to respond faster even if the cached entry can't be validated and might be stale!
- data centers should weaken consistency for faster response

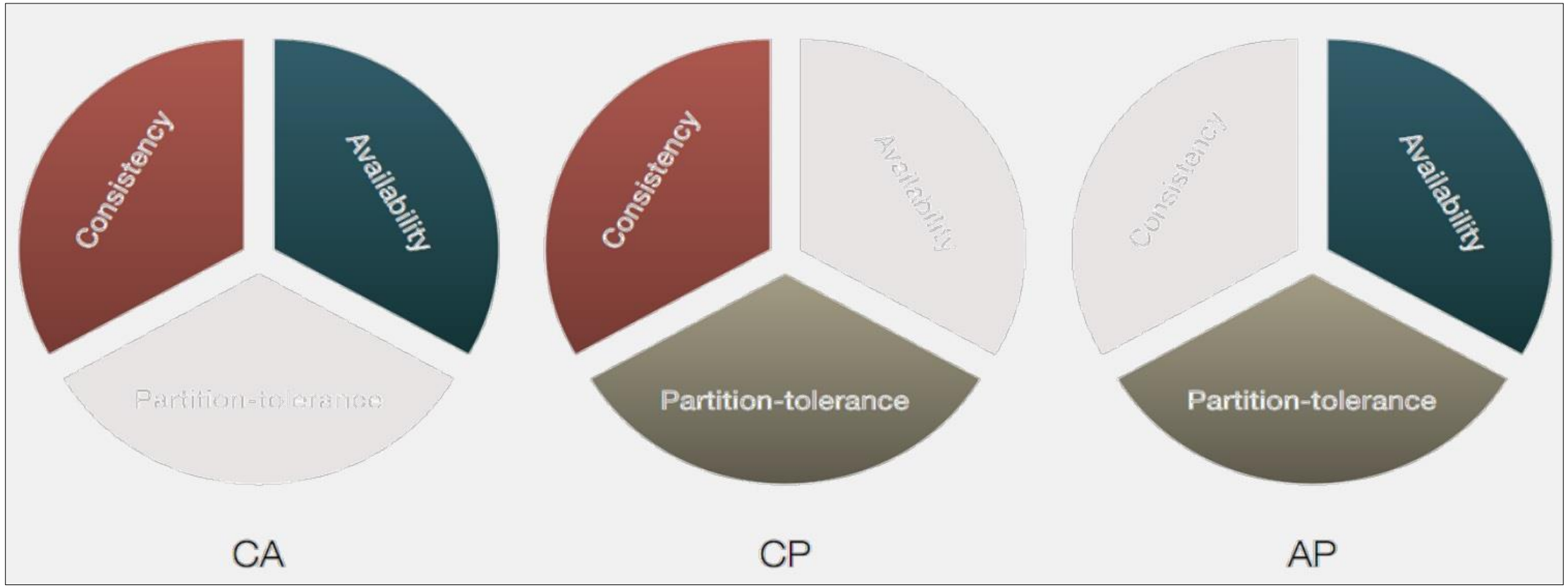
BREWER'S CONJECTURE

- started as conjecture, "proven" in 2002, became a theorem, but some researchers still argue that the "proof" is incomplete
- the CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:
  - consistency – all nodes see the same data at the same time
  - availability – a guarantee that every request receives a response about whether it was successful or failed
  - partition tolerance – the system continues to operate despite arbitrary message loss or failure of part of the system

DEFINITION


∃ total order
∀ operations
so that they look
as if they were
completed at a
single instant
Atomic/Linearizable
Consistency
Availability
every request
received by a
non-failing
node must result
in a response
Partition-tolerance
no set of failures
less than total network
failure is allowed to
cause the system to
respond incorrectly

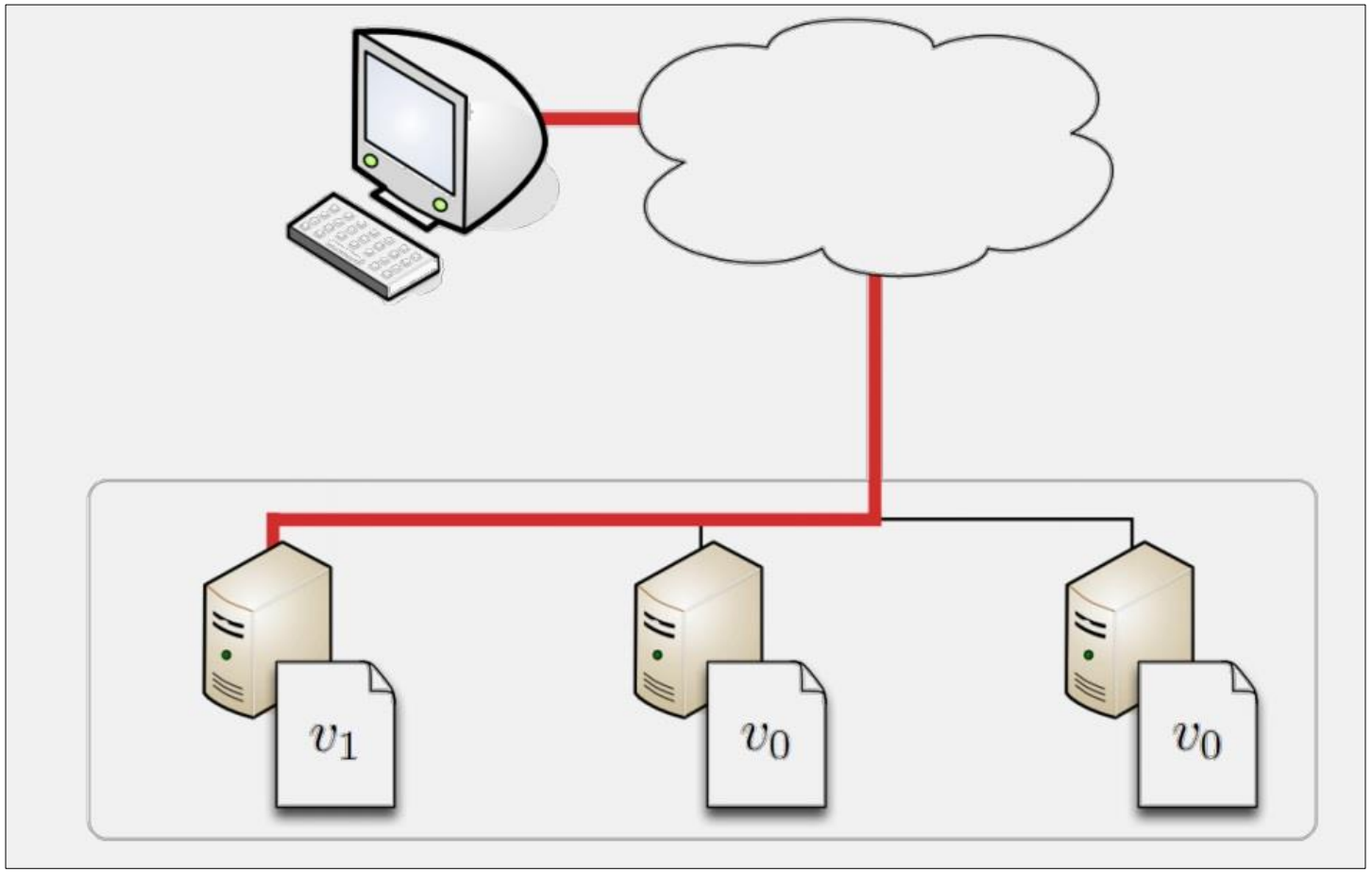
CAP PROPERTIES



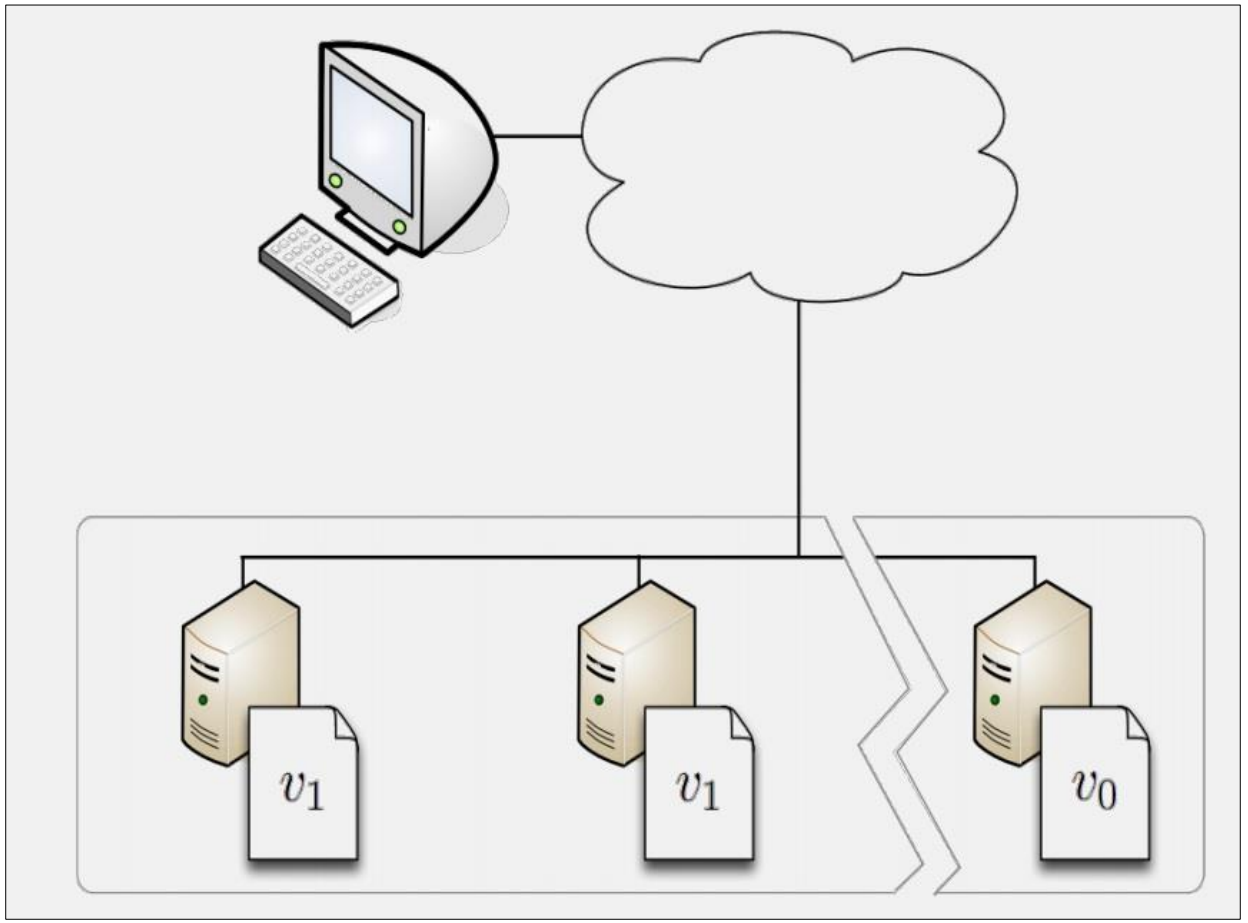
RESULTS FROM ASYNCHRONOUS SYSTEMS

- it is impossible in the asynchronous network model to implement a read/write data object that guarantees
  - availability
  - atomic consistency
- in all fair executions (including ones in which messages are lost)
- asynchronous – there is no clock – decisions made with local computations or messages

RESULTS FROM ASYNCHRONOUS SYSTEMS



RESULTS FROM ASYNCHRONOUS SYSTEMS



PARTIALLY SYNCHRONOUS MODEL

- allow each node in the network to have a clock, and all clocks increase at the same rate
- can use clocks as timers to know whether the message was received
- again, it is impossible in the partially synchronous network model to implement a read/write data object that guarantees the following properties:
  - availability
  - atomic consistency

HOWEVER…

- there are partially synchronous algorithms that will return atomic data when all messages in an execution are delivered (no partitions) – and will only return inconsistent data when messages are lost
- example: use a centralized server to receive all messages an have server respond to message to all nodes. Use time outs. If time outs fail to predict completion of successful message transfer, then atomic consistency may be violated.

T-CONNECTED CONSISTENCY

- if no partitions, then system is consistent
- if partition, then system can return stale data
- once a partition heals, there is a time limit (clock) on how long the system will take to return to consistency

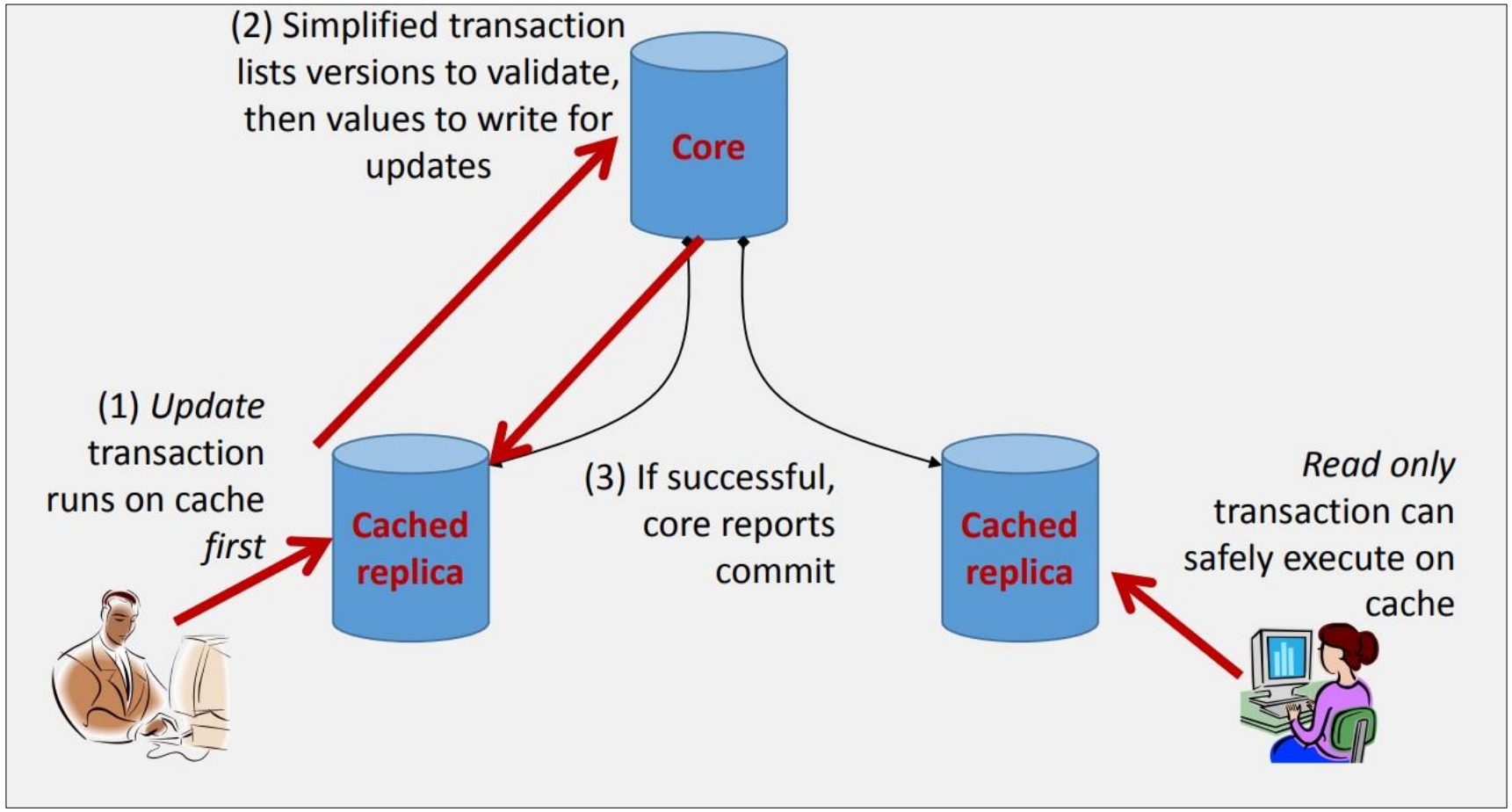
IS INCONSISTENCY A BAD THING?

- how much consistency is really needed in the first tier of the cloud?
  - think about YouTube videos Would consistency be an issue here?
  - what about the Amazon ‘number of units available’ counters. Will people notice if those are a bit off?

CONSISTENCY: TWO 'VIEWS'

- client sees a snapshot of the database that is internally consistent and "might" be valid
- internally, database is genuinely consistent, but the states clients saw aren't tracked and might sometimes become invalidated by an update
- inconsistency is tolerated because it yields such big speedups, although some clients see "wrong" results

A PICTURE OF HOW THIS WORKS



CONSISTENCY TRADE-OFFS

- CAP is a warning that strong properties can easily lead to slow services
- weak properties are often a successful strategy that yields a good solution and requires less effort

THE ACID MODEL

- a model for correct behavior of databases
- name was coined in the 60's
  - **atomicity** – even if "transactions" have multiple operations, does them to completion (commit) or rolls back so that they leave no effect (abort)
  - **consistency** – a transaction that runs on a correct database leaves it in a correct ("consistent") state
  - **isolation** – it looks as if each transaction ran all by itself. Basically says "we'll hide any concurrency"
  - **durability** – once a transaction commits, updates can't be lost or rolled back

TRANSACTIONS ARE THE BASIS OF ACID

- applications are coded in a stylized way
  - begin transaction
  - perform a series of read, update operations
  - terminate by commit or abort

```
Begin
      let employee t = Emp.Record("Tony");
      t.status = "retired";
                  customer c: c.AccountRep = "Tony"
                              c.AccountRep = "Sally"
Commit;
```

- system executes this code in an all-or-nothing way

WHY IS ACID HELPFUL

- developer doesn't need to worry about a transaction leaving some sort of partial state
  - e.g. showing Tony as retired and yet leaving some customer accounts with him as the account rep
- similarly, a transaction can't glimpse a partially completed state of some concurrent transaction
  - eliminates worry about transient database inconsistency that might cause a transaction to crash
  - analogous situation – thread A is updating a linked list and thread B tries to scan the list while A is running

THIS MOTIVATES BASE

- proposed by eBay researchers
  - found that many eBay employees came from transactional database backgrounds and were used to the transactional style of “thinking”
  - but the resulting applications didn’t scale well and performed poorly on their cloud infrastructure
- goal was to guide that kind of programmer to a cloud solution that performs much better
  - BASE reflects experience with real cloud applications
  - “Opposite” of ACID

BASE IS A METHODOLOGY

- BASE involves step-by-step transformation of a transactional application into one that will be far more concurrent and less rigid
  - but it doesn't guarantee ACID properties
  - argument parallels (and actually cites) CAP: they believe that ACID is too costly, and often, not needed
  - BASE stands for "**Basically Available Soft-State Services with Eventual Consistency**"

BASE TERMINOLOGY

- **basically available** – fast response even if some replicas are slow or cashed
- **soft-state service** - no durable memory
  - can’t store any permanent data
  - restarts in a “clean” state after a crash
  - to remember data, either replicate it in memory in enough copies to never lose all in any crash, r pass t to some other service that keeps “hard state”
- **eventual consistency** – OK to send “optimistic” answers to the external client
  - could use cached data
  - could guess at what the outcome of an update will be
  - later, if needed, correct any inconsistencies in an offline cleanup activity

BASE VS. ACID

- ACID code often much too slow, scales poorly, and end-user waits a long time for responses
- with BASE
  - code itself is way more concurrent, hence faster
  - elimination of locking, early responses, all make end-user experience snappy and positive
  - but we do sometimes notice oddities when we look hard
    - slightly different bidding histories shown to different people shouldn't hurt much, if this makes eBay 10x faster
    - upload a YouTube video, then search for it -> you may not see it immediately

REDIS: REMOTE DICTIONARY SERVER

- open source
- written in C
- data model is a dictionary which maps keys to values
- supports not only strings, but also abstract data types:
  - lists of strings
  - sets of strings (collections of non-repeating unsorted elements)
  - sorted sets of strings (collections of non-repeating elements ordered by a floating-point number called score)
  - hashes where keys and values are strings

REDIS: REMOTE DICTIONARY SERVER

- ultrafast response time
  - everything is in memory
  - non-blocking I/O
  - 100000+ read/writes per second
- periodically checkpoints in-memory values to disk every few seconds
- in case of failure, only the very last seconds' worth of key/values are lost

POTENTIAL USES

- session store
  - one (or more) sessions per user
  - many reads, few writes
  - throw-away data
  - timeouts
- logging
  - rapid, low latency writes
  - data you don’t care that much about
  - not that much data (must be in-memory)

GENERAL CASES

- data that you don’t mind losing
- records that can be accessed by a single primary key
- schema that is either a single value or is a serialized object

SIMPLE PROGRAMMING MODEL

- GET, SET, INCR, DECR, EXISTS, DEL
- HGET, HSET, KEYS, HDEL
- SADD, SMEMBERS, SREM
- PUBLISH, SUBSCRIBE

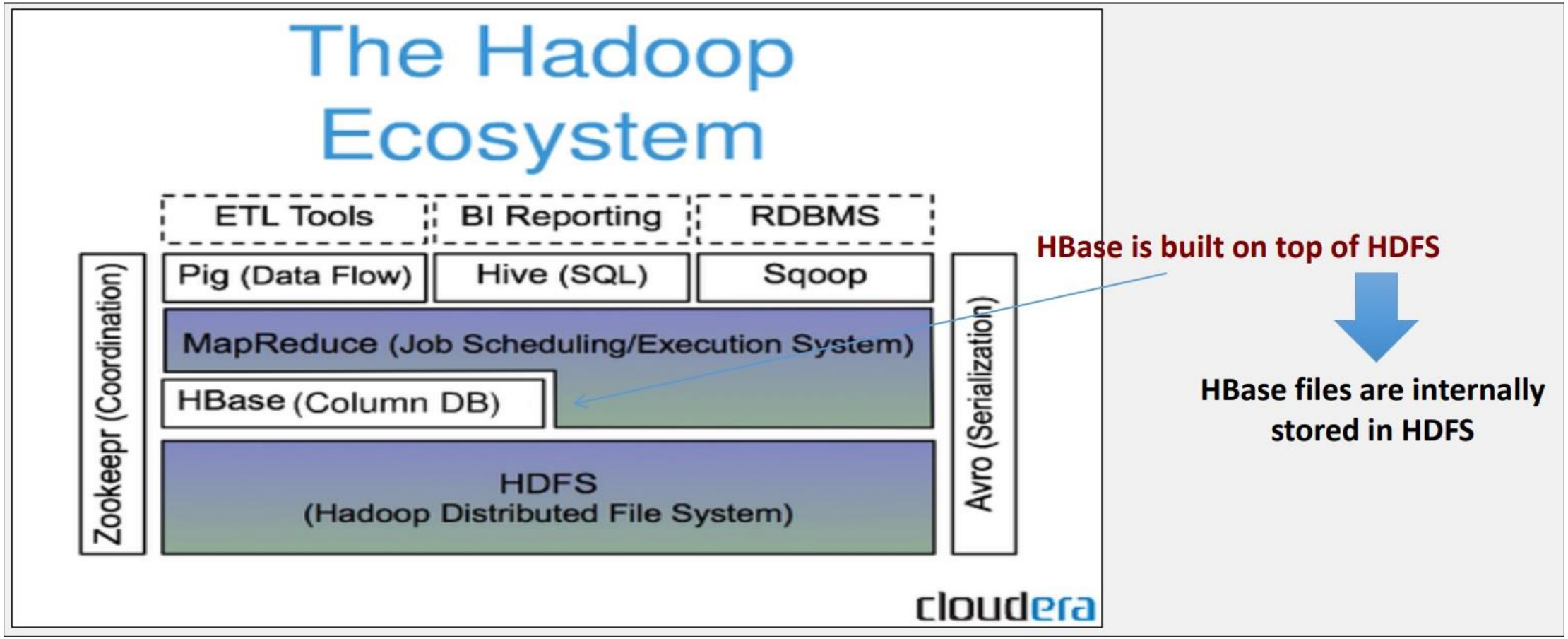
SUMMARY

- redis is not a database
  - it complements your existing data storage layer
  - e.g. stackoverflow uses redis for data caching
- publish/subscribe support

HBASE OVERVIEW

- HBase is a distributed column-oriented data store built on top of HDFS
- HBase is an Apache open source project whose goal is to provide storage for Hadoop Distributed Computing
- data is logically organized into tables, rows, and columns

HBASE – PART OF HADOOP'S ECOSYSTEM

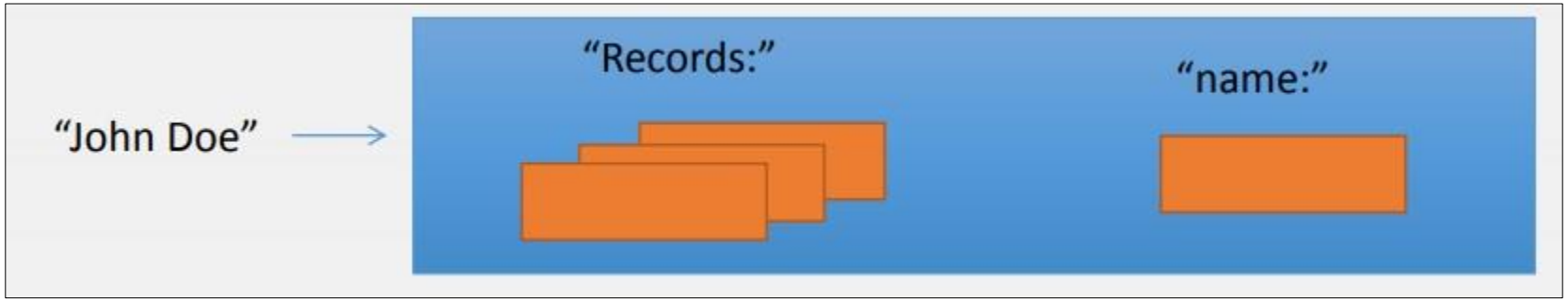


HBASE VS. HDFS

- HDFS is good for batch processing (scans over big files)
  - not good for record lookup
  - not good for incremental addition of small batches
  - not good for updates
- HBase addresses the above points
  - fast record lookup
  - support for record-level insertion
  - support for updates

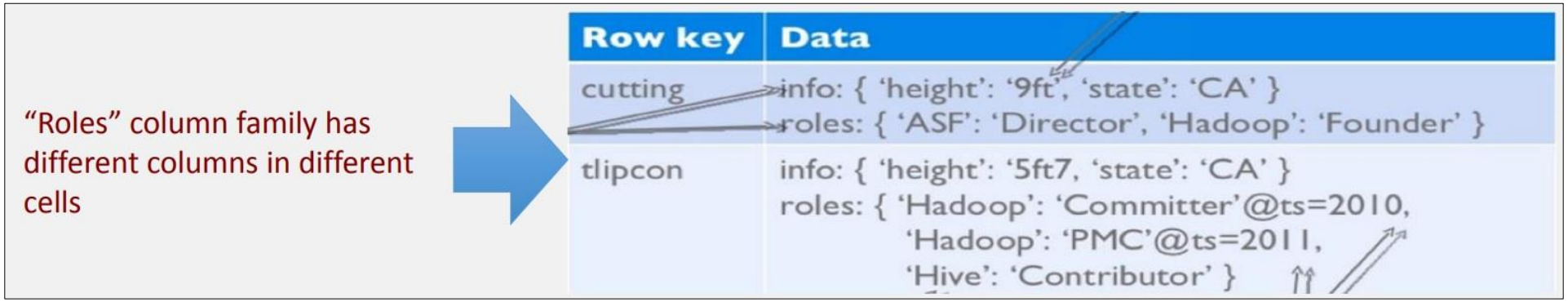
DATA MODEL

- a table in Bigtable is a sparse, distributed, persistent multidimensional sorted map
- map indexed by a row key, column key, and a timestamp
- supports lookups, inserts, deletes



NOTES ON DATA MODEL

- HBase schema consists of several **Tables**
- each table consists of a set of **Column families**
  - columns are not part of the schema
- HBase has **Dynamic columns**
  - because column names are encoded inside the cells
  - different cells can have different columns



NOTES ON DATA MODEL

- the **version number** can be user-supplied
  - does not have to be inserted in increasing order
  - version numbers are unique within each key
- table can be very sparse
  - many cells are empty
- **keys** are indexed as the primary key

Row Key	Time Stamp	ColumnFamily contents	ColumnFamily anchor
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"
"com.cnn.www"	t6	contents:html = "<html>..."	
"com.cnn.www"	t5	contents:html = "<html>..."	
"com.cnn.www"	t3	contents:html = "<html>..."	

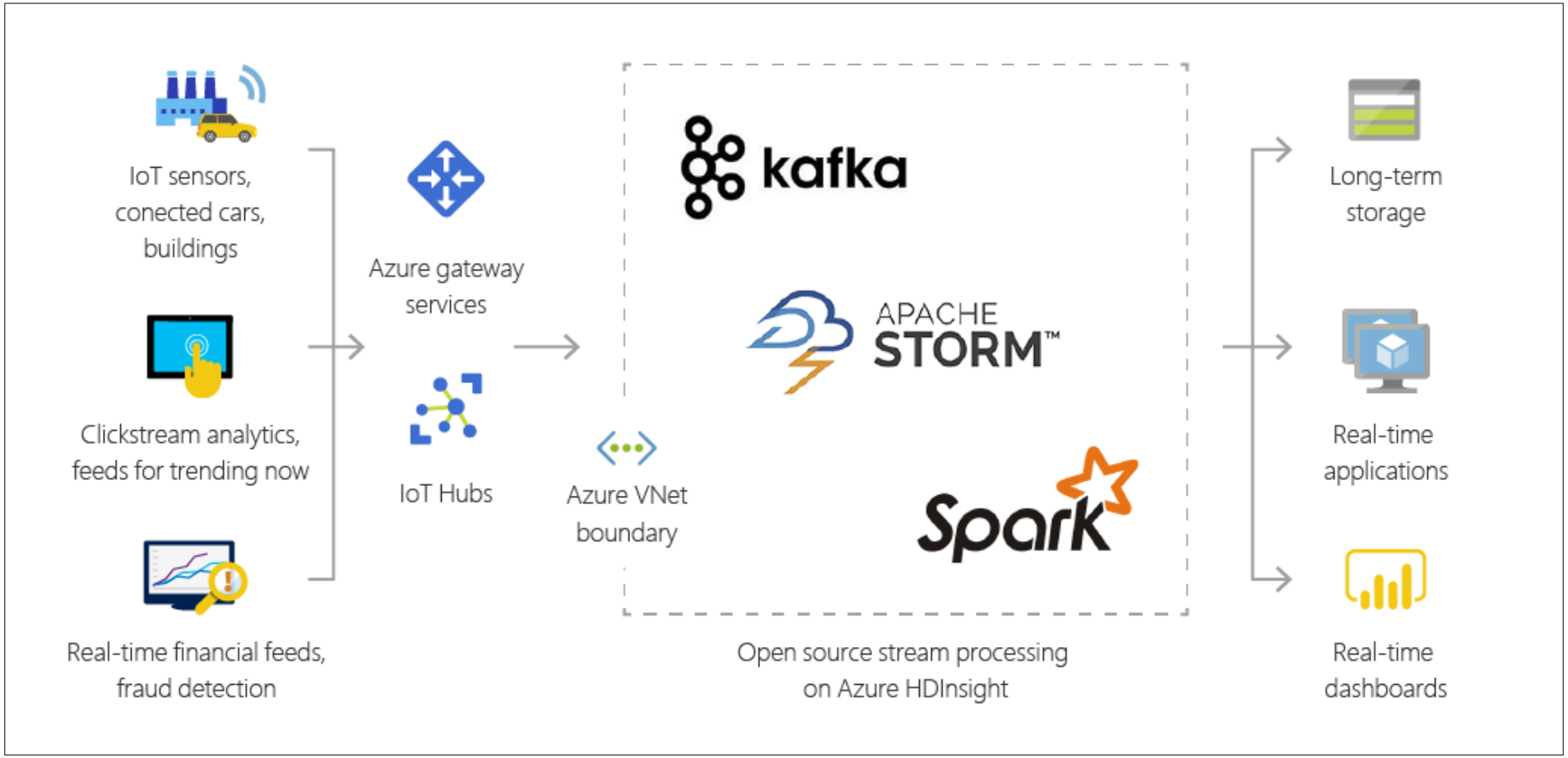
ROWS AND COLUMNS

- rows maintained in sorted lexicographic order
  - applications can exploit this property for efficient row scans
  - row ranges dynamically partitioned into tablets
- columns grouped into column families
  - column key = family:qualifier
  - column families provide locality hints
  - unbounded number f columns

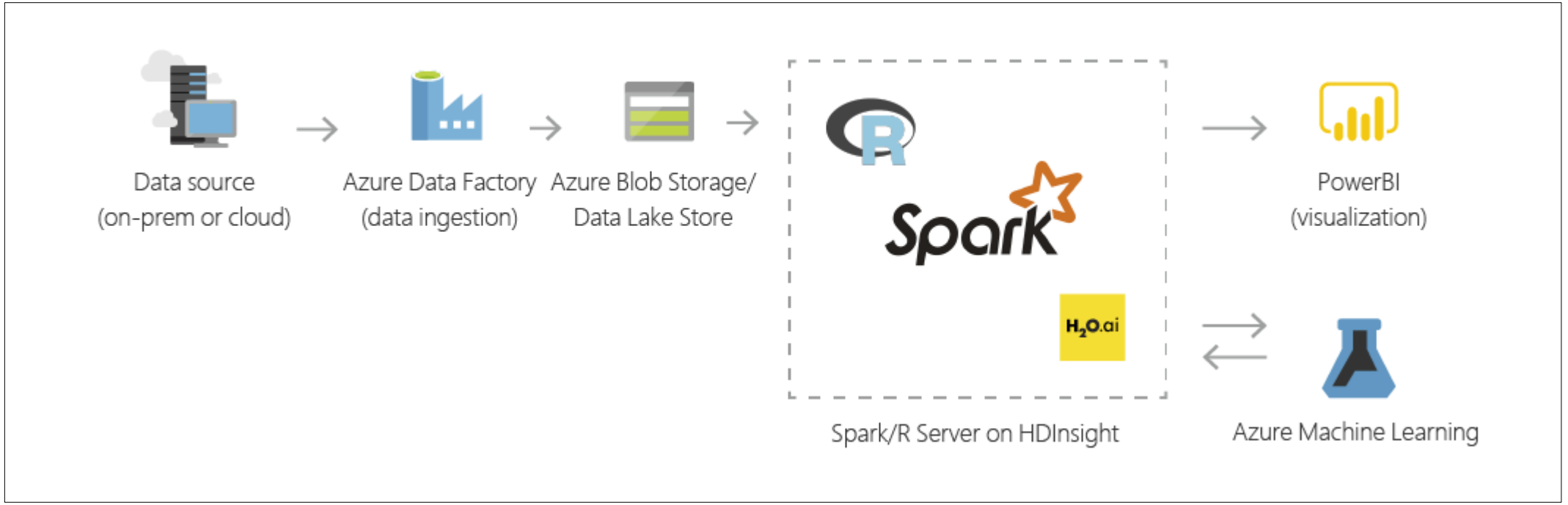
MICROSOFT AZURE HDINSIGHT

- managed open source big data analytics service for the enterprise
- users can create optimized clusters for Hadoop, Spark Hive, Hbase, Storm, Kafka, …

INTERNET OF THINGS



DATA SCIENCE AND MACHINE LEARNING



DATA WAREHOUSING