Lecture 2

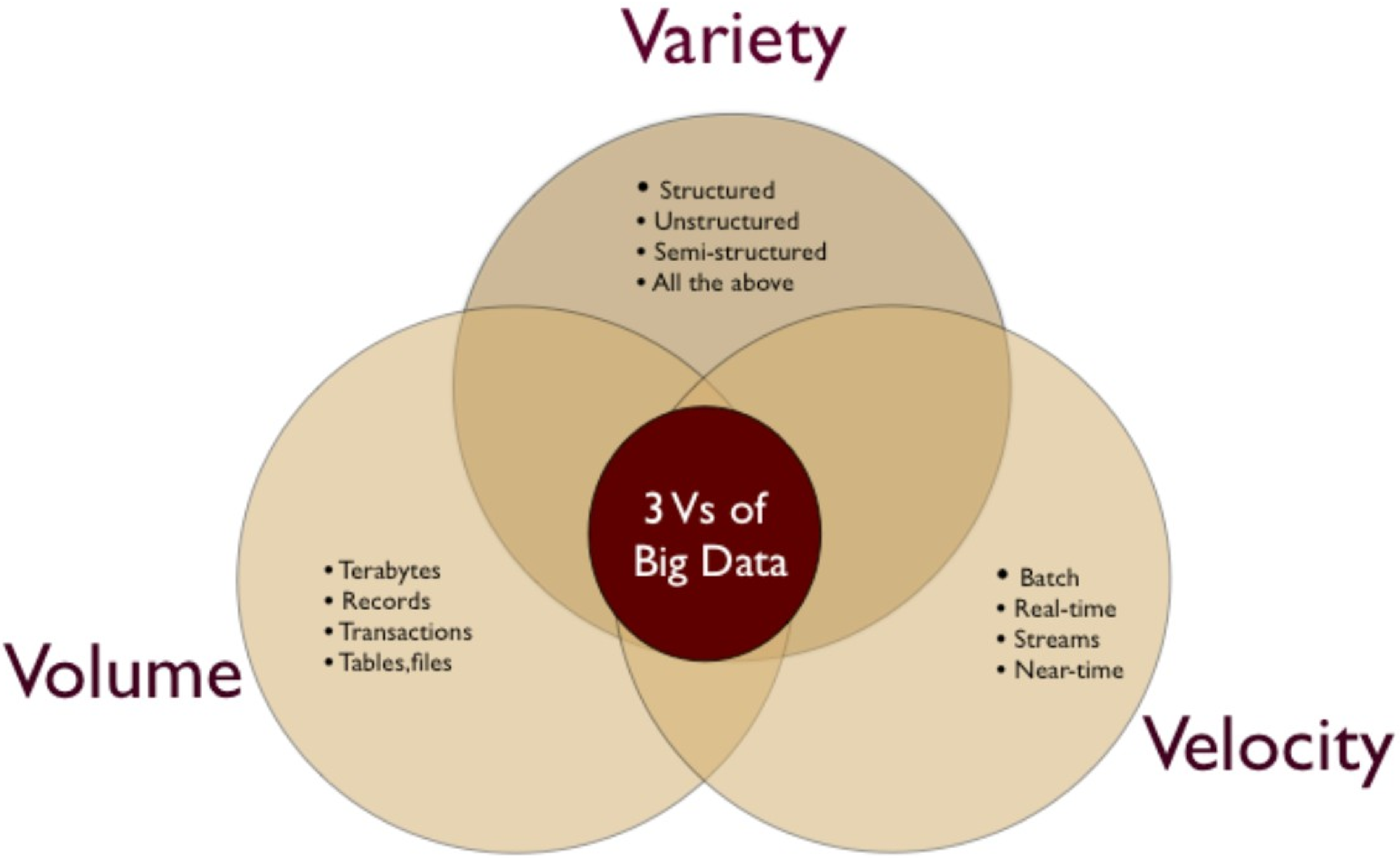
An Overview of Parallel Computing

3V of Big Data

Big Data is often defined using the following three V's:

- **Volume:** The dataset grows too large to be processed in the classical computing model
- **Velocity:** The data is arriving faster than a common server can process.
- **Variety:** The data comes in with great variations in structure and quality (semi-structured, unstructured, text)

These three Vs should exist at the same time.


Variety
• Structured
• Unstructured
• Semi-structured
• All the above
Volume
• Terabytes
• Records
• Transactions
• Tables,files
3 Vs of Big Data
• Batch
• Real-time
• Streams
• Near-time
Velocity

Consequences

Volume

- As the volume is getting larger than a single computing unit can handle, the data needs to get ***split into chunks*** that are small enough to be handled.
- When processing data with high volume on a set of computers, the ***communication of data*** is a typical bottleneck, related to the *Velocity*
- Algorithms must be designed to work on partial data, allowing for aggregating results.
- Active communication between different computers processing different chunks can become mandatory.

Consequences

Velocity

- In order to handle the speed, data is going to be ***fed into various nodes***
- A ***consistent view*** of the incoming and existing data is impossible to generate in real time.
- Nodes need to communicate with each other in an ***application-dependent manner***
- It is often impossible to store the incoming data, hence, a careful design of what to hold and what to forget is important

Consequences

Variety

- Data processing should not expect anything about the data in terms of
  - Format
  - Quality
  - Meaning
- ***General preprocessing of data*** is impossible (Velocity) or ineffective (Volume)
- ***Application-dependent preprocessing*** must be defined and implemented

Summary

- **Big Data** is a situation in which three aspects of Volume, Velocity and Variety hinder the application of classical approaches including
  - Relational Database Management Systems
  - Single PC / Server Solutions
  - Typical Input, Processing, Output pipeline

?-V of Big Data

- There are many other Vs that have been defined by various people including many aspects.
- These, however, are typically aspects of working with big data and not defining properties of big data.

10 V

- **Volume**: = lots of data
- **Variety**: = complexity, thousands or more features per data item, the curse of dimensionality, combinatorial explosion, many data types, and many data formats.
- **Velocity**: = high rate of data and information flowing into and out of our systems, real-time, incoming!
- **Veracity**: = necessary and sufficient data to test many different hypotheses...
- **Validity:** = data quality, governance, master data management (MDM).
- **Value**: = the all-important V, characterizing the business value, ROI, and potential of big data to transform your .
- **Variability**: = dynamic, evolving, spatiotemporal data, time series, seasonal, and any other type of non-static behavior in your data sources, customers, objects of study, etc.
- **Venue**: = distributed, heterogeneous data from multiple platforms, from different owners' systems, with different access and formatting requirements, private vs. public cloud.
- **Vocabulary**: = schema, data models, semantics, ontologies, taxonomies, and other content- and context-based metadata that describe the data's structure, syntax, content, and provenance.
- **Vagueness**: = confusion over the meaning of big data (Is it Hadoop? Is it something that we've always had? What's new about it? What are the tools? Which tools should I use? etc.) Note: I give credit here to Venkat Krishnamurthy (Director of Product Management at YarcData) for introducing this new "V" at the Big Data Innovation Summit in Santa Clara on June 9, 2014.

https://mapr.com/blog/top-10-big-data-challenges-serious-look-10-big-data-vs/

Introduction to Big Data

A Distributed Computing Approach

Transactions

For consistent and easy services, we want to be able to understand, that some operations happen exactly once (e.g., incrementing a count) with guarantees on the state of the sytem in case of success or error.

A **transaction** is a section of execution (program, SQL, data modification) that is protected in the sense that it **either fails leaving the system in a state as if the transaction never happened** or is **executed exactly once.**

Usually, we refer to the two properties as **transactional guarantees.** There is a third guarantee often assumed called „Atomicity“, which we introduce later together with threads.

SQL Example

```
BEGIN TRANSCATION
DELETE FROM Lecture WHERE StudentID=13
INSERT INTO Alumni (…) VALUES (...)
COMMIT
```

- This binds together both commands such that either **both** are successfully executed or **none**. From an application point of view, this means, that student 13 is deleted only if we were able to store him as an Alumni.

Example: 2-Phase-Commit

In order to guarantee transactions over distributed systems, we need to temporarily execute the transaction in our local system (Phase 1). If it already fails, we need to rollback. Then, we need to ask (Phase 2) all other nodes, whether they can accept the transaction. If at least one node disagrees, we need to rollback. Otherwise, we can make the transaction persistent.

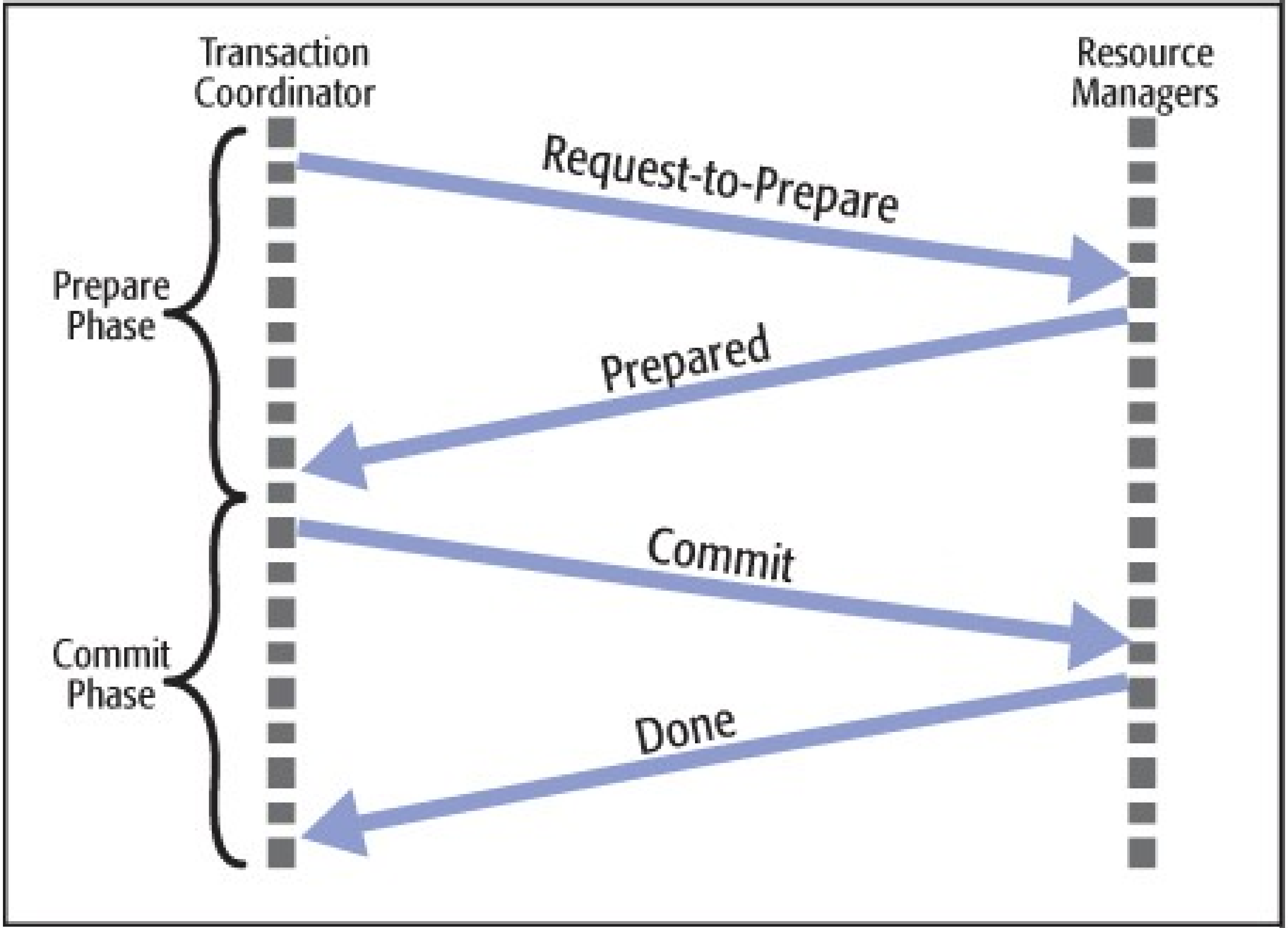


Figure 1 • The two-phase commit protocol

Some Observations for 2PC

- The system is in **inconsistent state** between Phase 1 and Phase 2: On the node issuing the transaction, the transaction has been performed while it has not been performed on other nodes. This can result in severe problems from an application perspective.

  Example: A shop has only one article left in stock. On two different nodes, the article gets independently reserved (Phase 1). What happens with Phase 2?

Some Observations with 2PC

- If a **single node fails**, it cannot agree to transactions and the complete system stalls.

  This leads to the development of systems, which replace the all-vote with a majority-vote, which leads to better scalability, but, in fact, no transactional guarantees in the strong sense.

  Note, however, that majority voting is also quite hard and communication-intensive.

Transactions over Big Data

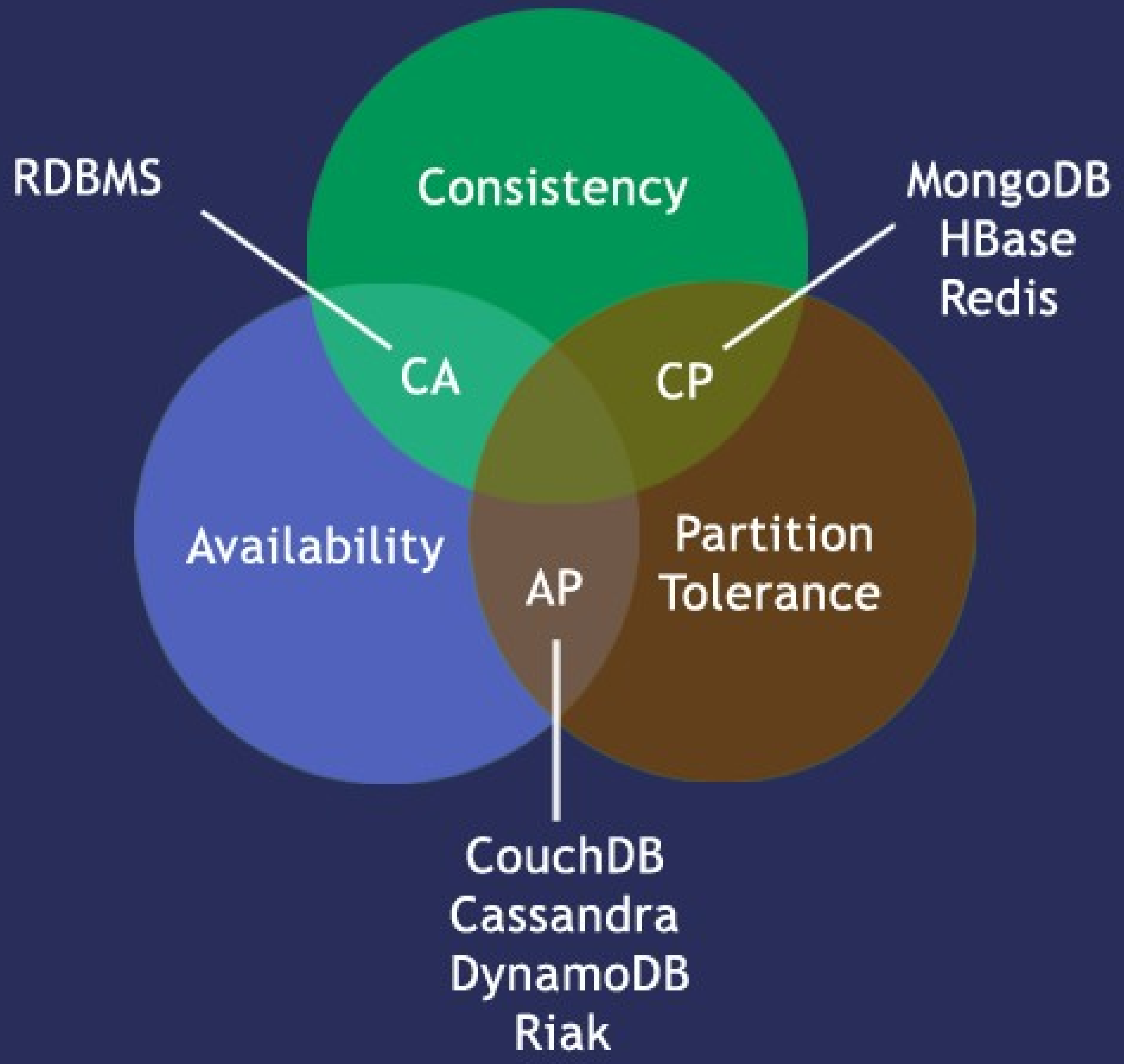
- When scaling over more than a few compute nodes, the system must expect and tolerate outages, which leads to the conclusion that **strong transactional guarantees are impractical** over big data

  Imagine an airplane with one motor. If the motor fails, the airplane will crash. If an airplane has 2 motors, we should try to design it such that it can also fly with one motor, right? If it had 1000 motors? How long would it take until at least one motor fails?

The CAP Theorem

- There is a central result in distributed systems, called CAP theorem, which states that the following three properties cannot hold at the same time for any distributed system:
  - Consistency: The view of data is the same anywhere in the distributed system
  - Availability: The system is available at any time
  - Partition tolerance: If the network fails and the system gets partitioned into pieces, it does not fail

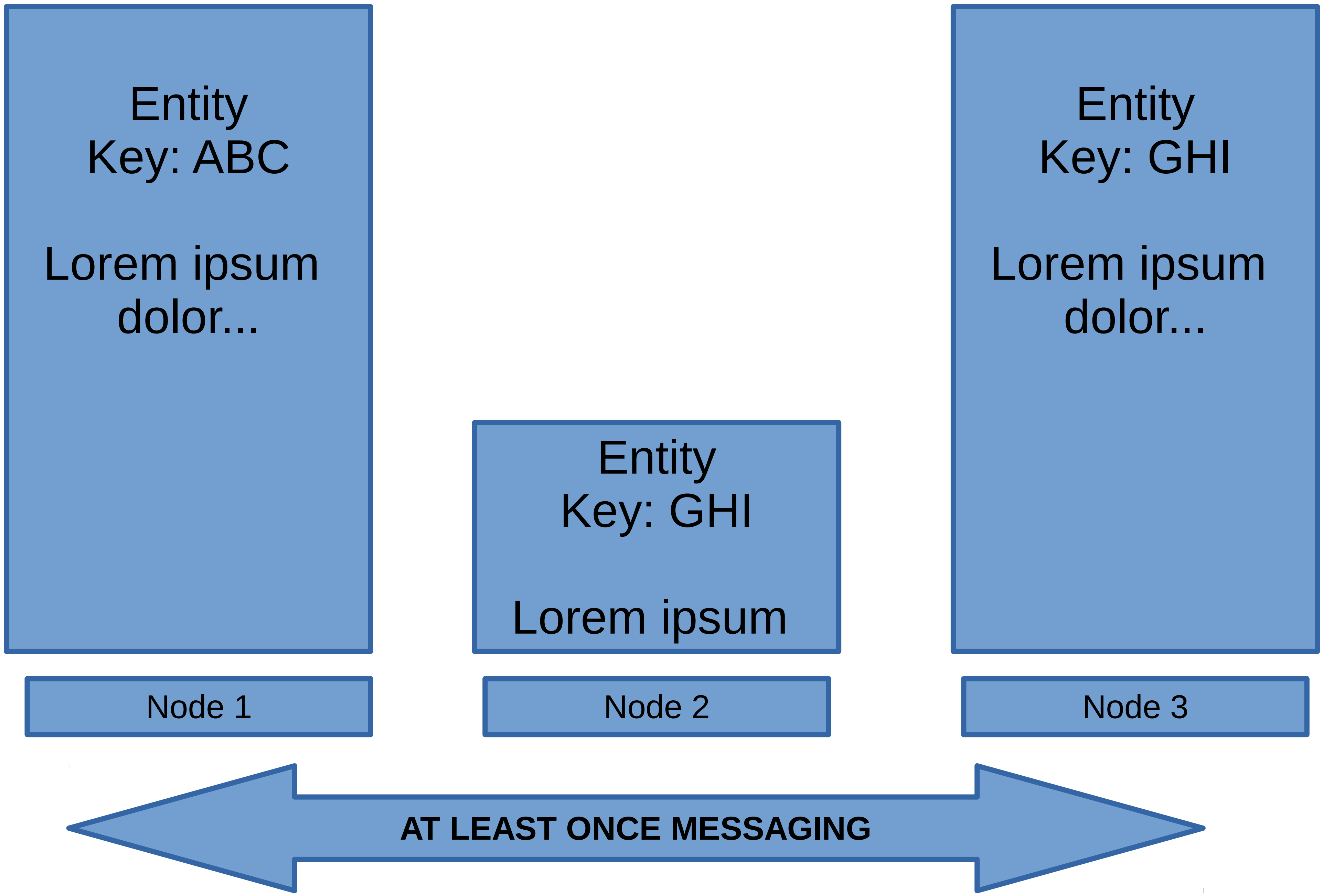
CAP Theorem


RDBMS
Consistency
MongoDB
HBase
Redis
CA
CP
Availability
AP
Partition Tolerance
CouchDB
Cassandra
DynamoDB
Riak

A simple model

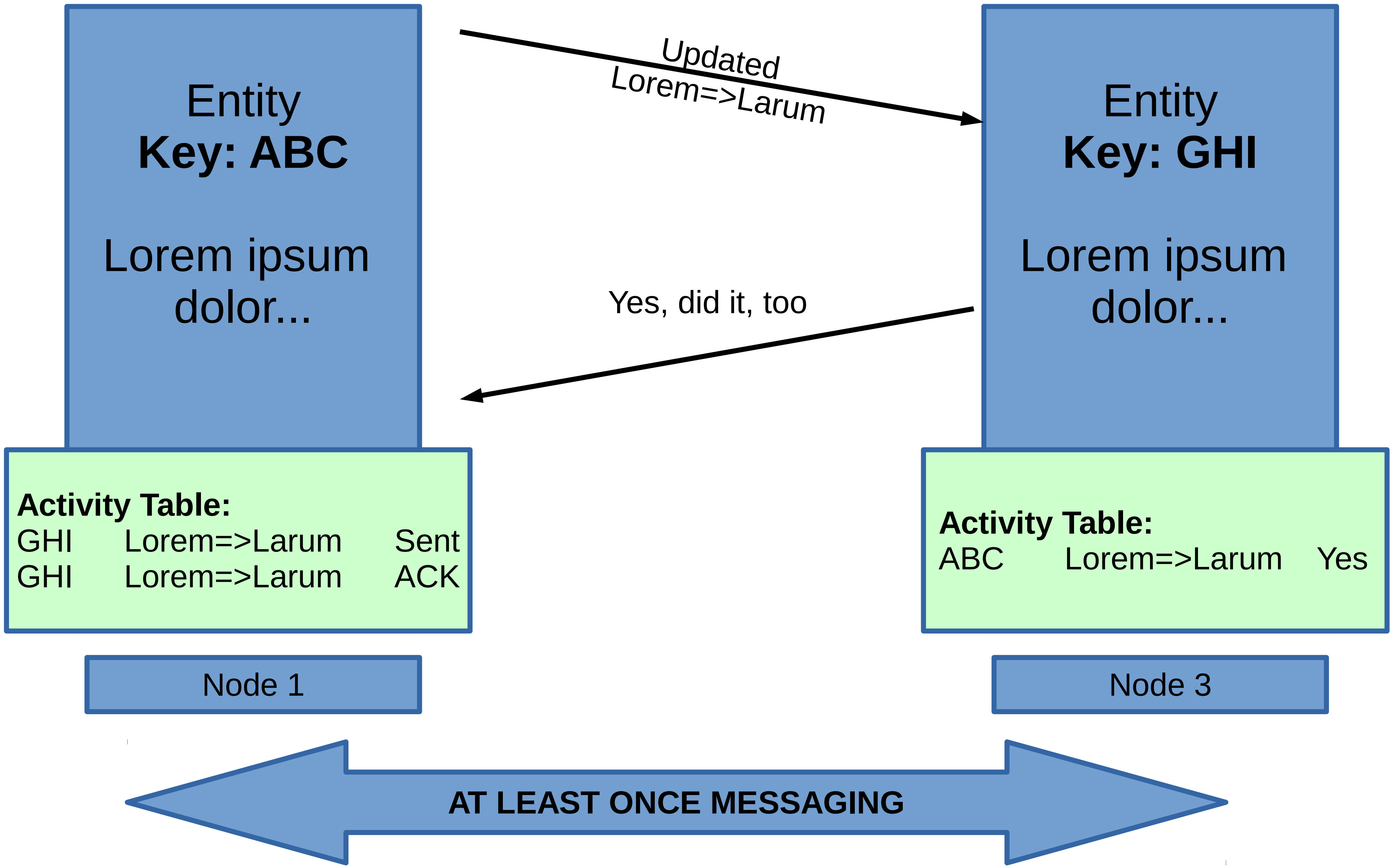
As transactions over large distributed systems are impractical, we need to define alternatives. But first, we give some names to our distributed system:

- An **entity** is a piece of data and/or code living on a single machine at a single time.
- Entities can **communicate** with each other. Communication is done via **Messages**, which are delivered **at-least once**
- Per-Partner state between entities is called **Activity**.


Entity
Key: ABC
Lorem ipsum dolor...
Entity
Key: GHI
Lorem ipsum
Entity
Key: GHI
Lorem ipsum dolor...
Node 1
Node 2
Node 3
AT LEAST ONCE MESSAGING

Assumptions

- **Entities** need to be globally named. This name will be called **key.** Entities can only be accessed by their name.
  If not, we would not be able to adress messages and understand the system from a local view.
- At-least once messaging implies **Activities**
  At least the processing implied from messages needs to be remembered. Otherwise, duplicate messages as well as out-of-order delivery lead to undefined behavior.
- Entities represent **disjoint sets of data.**


Entity
Key: ABC
Lorem ipsum dolor...
Updated Lorem=>Larum
Yes, did it, too
Entity
Key: GHI
Lorem ipsum dolor...
Activity Table:
GHI Lorem=>Larum Sent
GHI Lorem=>Larum ACK
Activity Table:
ABC Lorem=>Larum Yes
Node 1
Node 3
AT LEAST ONCE MESSAGING

Consequences

- **Transactions cannot span multiple entities**
  For example, we don't have the guarantee that two entities fit into one transactional scope (e.g., computer)
- *Hence, from a programmers point of view, an* ***entity is a transactional scope***
- **Messages are adressed to entities.**
  Otherwise, the application needs to know about the actual distribution including treatment of, for example, outages.

Activities and Per-Partner State

- Entities need per-partner state (Activities) as messages might be received more than once. We want to ensure:
  - The operation is performed exactly once
  - The answer should match the previously given answer
  - This needs state except for the case of idempotent operations.
- An operation is **idempotent**, if applying it twice does not change.
  - P(P(v)) = P(v).
  - Not changing anything or setting a value is trivially idempotent
  - Incrementing a value is obviously not idempotent, we need to construct an idempotent behavior, for example, with sequence numbers

Naming

- An entity can only have a **single consistent name**. Alternative namings are not consistent

  Students are identified via their Matrikel number. If we want to use, for example, their passport number as a second way, we have two choices. (1) We put it into the same transactional scope as the original data. Then, given a passport number, we need to scan over all entities to find the one, which contains the passport number. (2) We store the alternate index in its own set of entities and point to the name of the student entity. In this case, updates to the student are not consistent with the secondary index, as it is impossible to update the alternate index atomically. But, asynchronous messaging will help us in this case: We notify the secondary index for each update.

Messaging

- We have assumed at-least once delivery for messages. Sometimes, we need mechanisms to ensure at-most once processing of messages.
  - We could store the complete message and answer
  - We can also store something sufficient to remember, we have seen the message before.
  - We should have **essentially idempotent behavior,** for example, log entries are still allowed

Summary

In a scalable, distributed system, **entities** are introduced, which represent data that fits into one transactional scope. As transactions cannot span entities, **messaging** is used and partial inconsistency is accepted. As messages are not reliable, we need to **remember state** on a per-partner (entity) basis. This state is called **activity**. This approach implies, that only one **naiming scheme** is consistently applicable.

Section

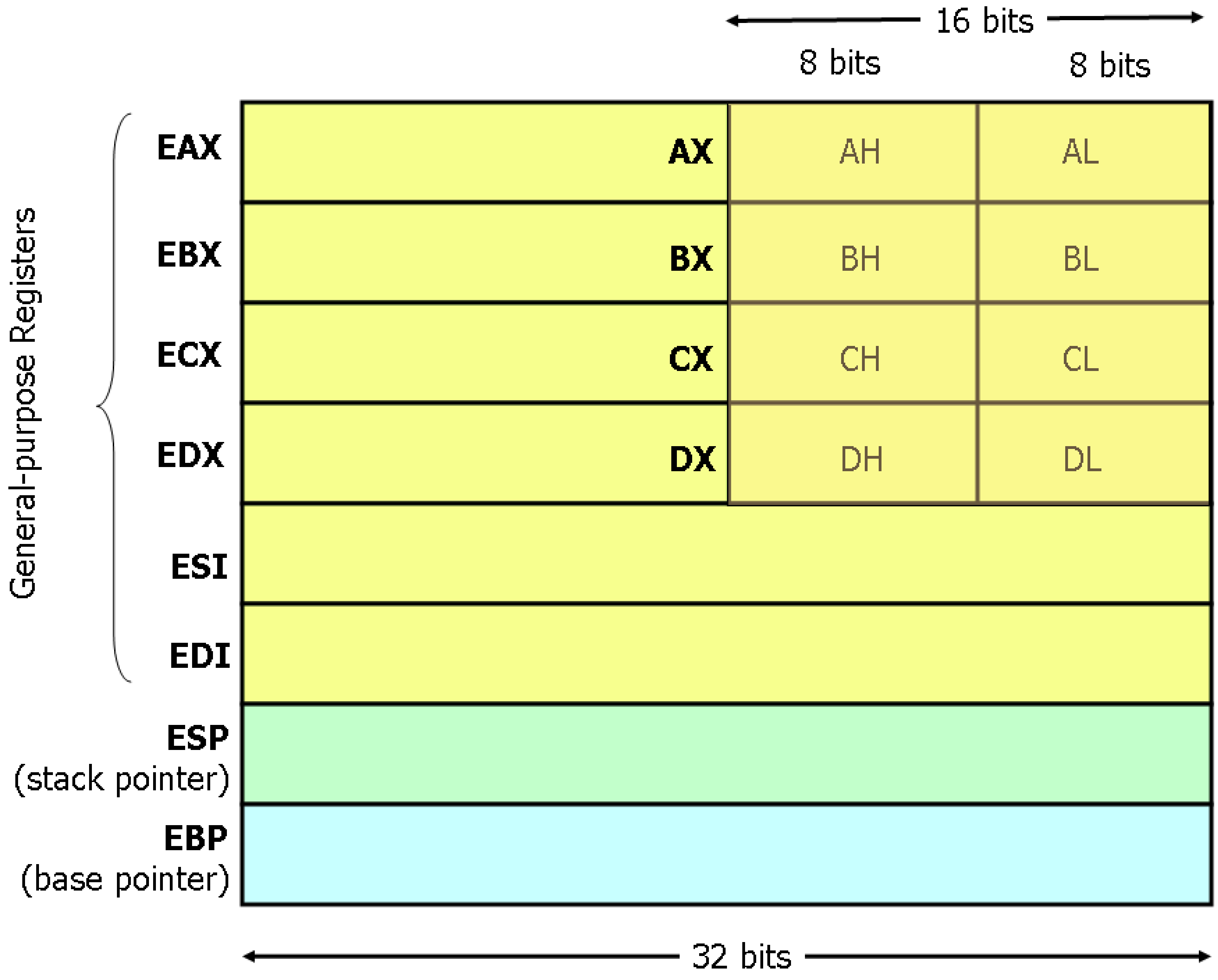
Processes & Threads

Algorithm

- An **algorithm** is an abstract sequence of actions that should be performed in order to solve a specific problem.
- A **computer program** is a formulation of an algorithm in a form that can be executed by a computer
- A **process** is a running instance of an program.

Programs and Computers

- A simple computer consists of
  - An Arithmetic Logical Unit **(ALU)**, which is capable of performing calculations between **registers**
  - A set of **registers** for holding values for calculations as well as
    - Memory pointers (Stack Pointer, Instruction Pointer, ...)
    - Flags (Overflow, Underflow, Zero, …)
  - Some amount of **Random Access Memory (RAM)** to hold
    - Binary programs
    - Data
  - Some **ports** to interface to the user
    - Keyboard
    - Terminal



http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Computers

- A computer basically works by the following loop
  - **Fetch** the next instruction from the main memory location given by the **Instruction Pointer (IP)** register.
  - Increment the IP
  - Execute the instruction

Computers

- This loop can be interrupted by a concept called **interrupt**
  - If the user presses a key, the hardware signals the CPU with an interrupt.
  - The execution jumps to a location called interrupt handler for further execution and jumps back to the current position (given by IP) after performing some operation based on the interrupt.

Instructions

- Usually, the CPU has a set of instruction from the following families
  - **Loading constants** instructions (loading constant values into registers)
  - **Moving** data between registers
  - **Computation** (with fixed registers)
  - **Loading and Storing** to and from main memory
  - **Port instructions** for interfacing external components
  - **Branching** instructions jumping based on results of previous computations (flags)
  - **Stack instructions** (push, pop, pusha, popa, call, ret)

Real-World: x86 Assembler

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

The x86 Assembler is simple and powerful. With a minimal of assembler commands, you can write complex programs including subroutines, recursion and more...

x86 is still supported by many modern computer systems and even different CPUs are similar.

Computers

- Timer Interrupts
  - Modern computers have **programmable timer interrupts** with which the CPU can be told to interrupt itself after a specific short amount of time calling an opertating-system-defined **interrupt routine**.
- Such timer interrupts enable the computer to perform **preemptive multi-tasking** in which the illustion of multiple running processes is generated**.**

Scheduling

- **Remember:** A process is a running instance of a program.
- The operating system **loads the executable** code into memory and creates a process data structure in a **global process table** holding the memory location of the program.
- With a timer interrupt, the operating system interrupts whichever process is actually executing and takes the **next process (round-robin scheduling)** from a list and lets it execute for a given amount of time.

Scheduling

- As the programmers do not know about other processes, each process assumes to „own“ the complete computer. This implies that
  - switchting between processes must protect all relevant registers and assumptions of the programmer
- Usually, this is called **context change**:
  - On timer interrupt, all registers are stored onto the stack (compare, for example, the x86 instruction **pusha**)
  - The stack pointer is stored in the **global process table**
  - The stack pointer of the next process is restored
  - The values of all registers are restored from the stack of the second process

Threads

While providing isolation and parallel processing (especially, when more than one CPU become available), processes have some limitations:

- A full context change has to protect a lot of data and, therefore, **takes a lot of time**.
- As processes **cannot share memory**, they also cannot share large data structures.

Therefore, a novel concept of a thread inside a process emerged.

Threads

A **thread** is – just like a process – an entry in the process table and, therefore, represents a stream of concurrent computations. However, multiple threads can exist inside a process and all threads share resources associated to the process. However, threads usually have their own stack.

Consequences:

- A context change between threads is – by an order of magnitude – **faster** than between processes
- Memory can be used **more efficiently** as compared to fork()-based copying of process memory, especially when operating on large datasets

Thread Risks

The shared resources of threads (e.g., memory, file descriptors) introduce severe problems, if not correctly used:

- Writing to data nearly parallel can **corrupt data**
- Writing to a terminal / console / file can **interfere**
- Locking can be used to mitigate those risks, however, introduces
  - **Possibly high overhead**
  - **Risk of dead-locks**

Thread APIs

- Threads have been introduced in cooperation with hardware vendors and – therefore – proprietary solutions preceded standardized solutions. Therefore, multi-threading has been a nightmare for long.
- C++11 gives the most modern standardization of threads from a high-level point of view. All other programming languages provide similar constructs, but sometimes in less sensible or less portable ways.

Major operations for threads

- **Create**
  Create creates a thread inside a process and usually takes the entry point as an argument. The entry point is usually a sub-routine.
- **Join**
  The current path of execution joins the threads path of execution. In fact, the calling thread waits for the given thread to finish
- **Detach**
  A thread is often tied to the current thread meaning that we can, for example, join the thread. We can also give up on this thread by calling detach. Then, the thread goes into an independent state.
- **Yield**
  The thread stops execution, but immediately gets rescheduled. This lets other threads execute for one time slice before the given thread is re-executed.

C++11 Thread Example

```
void foo()
{
  // do stuff...
}

void bar(int x)
{
  // do stuff...
}

int main()
{
  std::thread first (foo);     // spawn new thread that calls foo()
  std::thread second (bar,0);  // spawn new thread that calls bar(0)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

Java Example

```
class PrimeRun implements Runnable {
       long minPrime;
       PrimeRun(long minPrime) {
           this.minPrime = minPrime;
       }

       public void run() {
           // compute primes larger than minPrime
            . . .
       }
   }
   PrimeRun p = new PrimeRun(143);
   new Thread(p).start();
```

Alternatively:

```
   class PrimeThread extends Thread {
       long minPrime;
       PrimeThread(long minPrime) {
           this.minPrime = minPrime;
       }

       public void run() {
           // compute primes larger than minPrime
            . . .
       }
   }
```

Semaphores and Mutexes

Threads introduce new risks for application semantics, for example, when two threads consume the same resource at the same time.

- We need a mechanism to protect from parallel access to specific objects
- This is done through the concept of semaphores and mutexes

Mutual Exclusion

- An **atomic operation** is a CPU instruction that cannot be interrupted (is transactional)
  - Atomic operations need not be protected, because only one thread at a time will be successful
  - Most operations, however, are **non-atomic.** *For example, calculating the value of a variable and storing it into main memory is usually done in two (or more) instructions.*

Mutual Exclusion

- A **critical section** is a piece of a program that we want to protect in a way such that only one thread can be inside the section. Other threads may wait or differently handle the case that they fail to enter this section.
- In multi-threading situations, we can protect critical sections through the use of semaphores and mutexes.

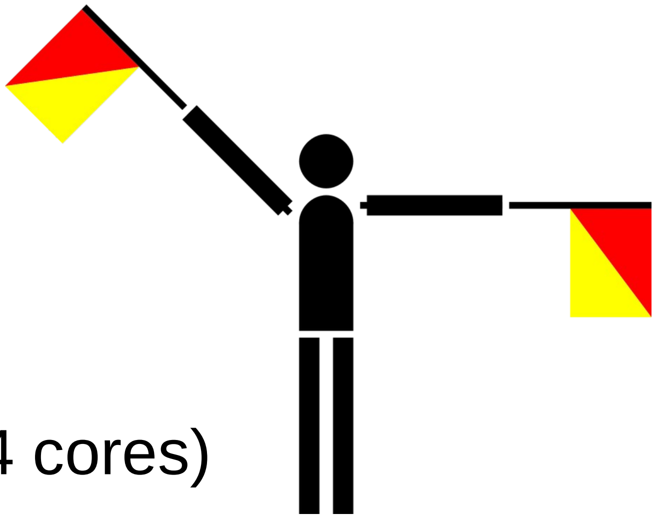
Semaphore

- A **semaphore** is a data structure holding a single counter and supporting two operations
  - **Decrement:** Decrease the counter
  - **Increment:** Increase the counter

  where the value is not allowed to take negative values. If a decrement would result in a negative value, the Decrement operation usually waits until someone else calles Increment.

  It is beyond the scope of this introduction to discuss, how semaphores can be implemented

Semaphore



- The main usecase of a semaphore is to assign a limited resource (e.g., one out of 4 cores) to services.
- A special case of a semaphore is a **mutex**, where Mutex stands for Mutual Exclusion):

  A mutex is a data structure with two states: locked or unlocked and two operations

  - Lock: Try to acquire a lock, usually waits in conflics
  - Unlock: Release a previously acquired lock for others

  Conceptually, this can be implemented as a binary semaphore...

Real-World: C++11

- A third operation **try_lock** is introduced, which immediately returns, if it fails to acquire the lock.
- Book-keeping with locks is tedious (think of thousands of global variables for all your locks and some typo somewhere), therefore, C++ introduces the following classes:
  - **std::unique_lock** takes care that a lock is released on destruction.
  - **std::lock_guard** protects even more: It releases the lock, for example, on exception
  - **std::scoped_lock** can lock more than one mutex at a time with deadline avoidance.

That three helper classes have been standardized highlights that the simple concept of a mutex seems to be extremely problematic in applications.

Summarizing Remarks

- Dealing with **parallel execution** is needed for big data.
- The concept of a **process** is simple and powerful, but cannot share enough resources
- The concept of a **thread** can be efficient, but it is not always easy to coordinate the conflicts between several threads.
- A mutex can be used to protect **critical sections**, however, it is not easy to apply

Annotation-based Parallelism

- One can even take the discussion one step further and claim that users of big data systems should not be exposed to parallel computing.
- While this is impossible, one step in this direction has successfully been done with OpenMP, a library for Shared-Memory Multiprocessing based on **source code annotations.**

Idea

- Given some **sequential** program, it usually contains simple patterns that can quite easily be parallelized. If we mark those patterns, we can push all the burden of parallization to the compiler.
- The most important patterns are **large loops**.

Example

```
[...]
vector<double> roots;
const size_t num_values = 1024*1024*256;
int main(void)
{
  roots.resize(num_values); // first allocate memory
  #pragma omp parallel for
  for (size_t i=0; i < roots.size(); i++)
  {
    roots[i] = sqrt(i);
  }
  cout << "Roots generated." << endl;
  return 0;
}
```

Example

- Such a program can compile and work sequentially.
- If, however, OpenMP is activated, the compiler will take care of parallelizing the for loop that was marked there.
- The #pragma statements for OpenMP can declare thread-local variables, scheduled and a lot more...

Example: Critical Section & Atomics

```
int main(void)
{
  cout << "Sequential processing." << endl;
  srand(time(NULL));
  size_t total_wait = 0;
  #pragma omp parallel
  {// This is run by each thread once.
      size_t wait_time = 1000000 + rand() % 2000000; // choose a random wait
time
      usleep(wait_time);
      #pragma omp atomic
      total_wait+=wait_time;
      #pragma omp critical
      cout << "Thread " << omp_get_thread_num() << " waited " << wait_time  <<
               endl << "Globally, we waited already " << total_wait << endl;
  };
  cout << "Now, all threads finished." << endl;
  return 0;
}
```

OpenMP: Risks

- With the vocabulary we have learnt so far, you can write quite efficient parallel programs with just some annotations. But beware that
  - All parallel sections **must not rely on the order of execution**!

  Typical problem: Appending values

  vector<...> result
  std::transform(data.begin(), data.end(), back_inserter(result), [](){...})

  While this would work, when protected as a critical section, the back_inserter would insert elements in arbitrary order and not in the order the programmer expected from sequential execution.

Summary

Key Words to know and understand:

3V of Big Data; Transactions, Consistency, CAP theorem; Entity and their Keys/ Transactional Scope; Messages; Activities and per-partner state; Idempotence

Algorithm, Program, Process, Preemptive Scheduling, Context Change, Thread, Mutex, Atomic operation, Critical Sections

OpenMP: Simple annotation scheme for atomics, critical section, and parallelism.