Big Data

Donald Kossmann & Nesime Tatbul
Systems Group
ETH Zurich
What we have learned so far...

- **What is Big Data?**
  - all the major buzzwords

- **Relational Databases Revisited**
  - data warehouses: architecture, star schema, SQL++
  - query processing 101, parallel query processing
  - algos for Cube, Top N, etc., column stores
  - (operational BI – deferred)

- **Hadoop**
  - Map Reduce programming model
  - overview of Hadoop eco system
Goal for today: Cloud Computing

• Get the infrastructure right
  – we need many machines
  – machines will store and process data
  – define interfaces to provision these machines

• Determine the right number of machines
  – goal: elasticity
  – (you don’t; automatically and magically get it right)

• For now, we stay in the relational world
  – show how it works for SQL (focus on OLTP)
Overview

• **What and why cloud computing?**
  – technologies and promises

• **Some examples**
  – Amazon Web Services

• **Databases in the Cloud**
  – transactions (many small reads & updates – OLTP)
What is Cloud Computing?

• Use of computing resources as a service
  – resources = software, platform, infrastructure
    • e.g., word processor, database system, CPU, disk
  – service: automate deployment of resource
    • e.g., start and end time, availability, SLAs, etc.

• Resources can be remote or local
  – actually, you typically do not care
  – you care about the what and when:
    • what kind of resource is used at what point in time
  – you do not care about the how and where
    • unless you have legal / compliance issues
Principles of Cloud Computing

• **Automation**
  – „program“ mundane IT tasks (e.g., backup, ...)
  – provide a REST / Web Service for these tasks

• **Virtualization**
  – decouple software from hardware
  – specify „what“ to deploy not „how“ or „where“

• **Pay-as-you-go**
  – rent software and hardware: do not buy!
Promises of Cloud Computing

• **Reduce Cost**
  – utilization of hardware and software (ideal is ~100%)
  – pay-as-you-go & efficient (no overheads)
  – no vendor lock-in

• **Reduce Time to Market**
  – focus on business problem (not IT)
  – no configuration, no provisioning, automatic security etc.
  – development framework (for enterprise Web apps)

• **Operating & Support (→ cost + time-to-market)**
  – SLAs: availability, guaranteed response times, ...
  – security
  – elasticity: scale-out and down with workload
  – multi-tenancy (support for SaaS)
Pain Point: Cost

• Failures
  – penalties for missed SLAs

• System administration
  – configuration & patches (security)

• Software licenses + maintenance
  – too many software layers

• Hardware
  – 20% vs. 90% utilization
  – fault-tolerance on cheap HW

• (Other: economy of scale)
  – Better optimization
  – Purchase volume, ...
Technical Background

• Many technical / business trends enabled the cloud
  – clusters: since 1990s
  – networks, Internet: since 1970s, boost “2000 Bubble”
  – virtualization: since 1960s
  – REST services: since 1990 (Web), since 2000 SOA
  – distributed computing: since 1980s
  – key value stores: since 2000s
  – ...

Business Trends

• **1940s: Can we do this?**
  – some things simply could not be done

• **2012: How can we best do this?**
  – everything can be done
  – many different metrics for “better”
    • cost, time-to-market, risk / robustness,
    • 99-percentile performance, data freshness, quality, ...
  – often good enough is good enough
  – sometimes better is better
## What to optimize?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Traditional</th>
<th>Cloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost [$]</td>
<td>fixed</td>
<td>optimize</td>
</tr>
<tr>
<td>Performance [tps, secs]</td>
<td>optimize</td>
<td>fixed</td>
</tr>
<tr>
<td>Scale-out [#cores]</td>
<td>optimize</td>
<td>fixed</td>
</tr>
<tr>
<td>Predictability [σ($)]</td>
<td>-</td>
<td>fixed</td>
</tr>
<tr>
<td>Consistency [%]</td>
<td>fixed</td>
<td>???</td>
</tr>
<tr>
<td>Flexibility [#variants]</td>
<td>-</td>
<td>optimize</td>
</tr>
</tbody>
</table>

*Put $ on the y-axis of your graphs!!!*  

[Florescu & Kossmann, SIGMOD Record 2009]
Key Enabling Technologies

- Clusters
- Computer Networks
- Virtualization
- Service-oriented Computing
Scale Up vs. Scale Out

• **What is better?**
  – 1 machine with 1000 cores:
    • bigger problem -> bigger machine
  – 1000 machines with 1 core
    • bigger problem -> more machines

• **It depends what you are looking for...**
  – performance: network within/across makes difference
  – scalability: get shared-memory to work?
  – cost: it depends
  – flexibility: 1000 machines with 1 core win

• **What matters today**
  – scalability and flexibility
  – design for scale out!!!
The Data Center of the Past

- one machine
  - many tubes
- one storage system
  - many tapes
- one terminal
- operated by humans
  - one role for all
The Data Center of Today
The Data Center of Today

• One building
  – somewhere where energy is cheap

• Many zones / clusters
  – each cluster with its own switch
  – each cluster with its own security

• Each cluster has many racks
  – each rack has a switch
  – each rack has many machines

• Each machine has many sockets, disks, MM
  – each socket with many cores
  – complex network within machine

• Roles: Administrators, Developers, Users
“When you set up a data center you have many problems. When you run a data center you have only one problem.”

[T. Roscoe, 2011]
Network

• The network is a hierarchy
  – within socket
  – within machine
  – within rack
  – within cluster
  – within data center
  – across data centers

• No scale-out of network yet
  – cannot replace big switch by many small ones
  – (active, ongoing topic for research)
  – need to partition data, shape traffic
Key Enabling Technologies

- Clusters
- Computer Networks
- **Virtualization**
- Service-oriented Computing
Virtualization

• **Principles**
  – decouple resource from service
  – share resources, dynamic provisioning, migration

• **Apply these principles at different levels**
  – software service: map URL to virtual machine
  – machines: map virtual machine to physical machine
  – storage: map key to block on physical machine (KVS)

• **Advantages of Virtualization**
  – increase utilization
  – improve fault tolerance
  – improve manageability
Virtual Machines

• **Idea: Decouple software from hardware**
  – provide a sandbox with OS and any software stack
  – move sandbox (=VM) if hardware overloaded

• **Advantage**
  – better utilization of hardware

• **History**
  – idea pioneered in the 60s for IBM mainframes
  – also applicable to desktops

• **Hypervisor**
  – dispatch VM calls to CPUs
  – monitor usage of VMs
  – move VMs if overloaded

• **But no fail-over!!!**
Logical Architecture of DB Apps

Client

HTTP

XML, JSON, HTML

Web Server

FCGI, ...

XML, JSON, HTML

App Server

SQL

records

DB Server

get/put

block

Store
Mapping Layers to Machines (Example)

Client → Store Area Network → Router → Client

A1 A2 A1 A3 A4 A5 A6 A6

DB1 ... DB6

Web Server
App Server
DB Server

Store Area Network
Mapping Layers to Machines

• There are many different ways to do that
  – we will revisit this point later

• Many different technologies at play
  – e.g., virtual machines (OS) to migrate “A”s and “DB”s
  – e.g., key-value store (DHT) to implement SAN

• Potential bottlenecks
  – router / network (we have talked about that)
  – database (we will talk about that)
  – everything else scales out nicely
Distributed Hash Tables (KVS)

• **Simple interface**
  – get(key)
  – put(key, value)  // two flavors: insert and/or replace

• **Principles**
  – consistent hashing, allows to extend / shrink “ring”
  – “finger tables”: support logarithmic access times
  – chain replication: support fault tolerance

• **Examples**
  – late 90s (research): Chord, Pastry, ...
  – mid 2000s (applied research, product): Cassandra, Dynamo, ...

• **Virtualizes storage layer: fault tolerant, elastic, fast**
  – decouples machine from “storage service”
Consistent Hashing

- Each machine can store several partitions
- Partitions should have even load -> machines have even load
Consistent Hashing

- Overload in one partition -> split that partition
- Split does not affect other partitions
Finger Tables

- Clients can send requests to any node in ring
- Nodes serve as routers within ring (logarithmic time and space)
Chain Replication

- Failure of one/two machines does not result in loss of data
- Many protocols to keep finger table, replication, etc. up to date
Why DHTs?

• They virtualize storage
  – client needs to know of only one node

• They tolerate failure in a cheap and simple way
  – e.g., no fancy technology for failure detection

• They are elastic
  – can add and remove machines at any point in time

• They are okay fast
  – logarithmic access guaranteed
  – caching at client improves situation significantly
Key Enabling Technologies

• Clusters
• Computer Networks
• Virtualization
• Service-oriented Computing
Services

• **Goal: Machine-to-machine Communication**
  – next step of automation
  – manage complex organizations

• **What is a service?**
  – a piece of code (~module) with an API
  – responds to messages, sends messages
  – autonomous, decoupled: may not respond, evolve
  – defines SLAs (non functional properties)
  – typically penalty model if violates SLA
Web Services: Standards

- **URI**
  - a way to give a name to a service
- **SOAP**
  - data format (XML for envelop and body)
  - specify bindings to protocols and intermediaries
- **WSDL**
  - define interface, message types, supported bindings
- **UDDI**
  - white, yellow, green pages of services (directory)
- **Or simply use REST**
  - URI + standardized http/https interface (get, post, ...)
Web Services & Cloud

• Web Service Call to start a VM
  – specify “small”, “medium”, “large”
  – possibly specify the image to “boot”
  – Web Service Call to end a VM

• Web Service Call to mount a (virtual) disk

• Web Service Call to “get / put” items in KVS

• Web Service Call to start a Hadoop job

• ...
Overview

• What and why cloud computing?
  – technologies and promises

• **Some examples**
  – Amazon Web Services

• Databases in the Cloud
  – transactions (many small reads & updates – OLTP)
Kinds of Clouds

• Functionality offered
  – SaaS: application (e.g., Salesforce)
  – PaaS: platform (e.g., database, queue)
  – IaaS: machine, storage, network

• Sharing of resources
  – public: open to everybody (e.g., Amazon)
  – private: behind firewall of organization (e.g., CS)
  – community: shared by set of organizations
IaaS

- **Goal:** Provide Computing Resources
  - (persistent) storage
  - CPU hours
  - network bandwidth
  - *IaaS plays the role of HW in the „new world“*

- **Straightforward way to implement principles**
  - E.g., REST API to create new storage objects
  - Virtualization: use XenSource, VMWare, ...
  - Pay-as-you-go: meter REST calls
Overview of Vendors

• Amazon Web Services (AWS)
  – IaaS and PaaS provider (public cloud)
  – high volume / low margin strategy

• Google AppEngine
  – PaaS with integrated IaaS (public cloud)
  – (may move into IaaS business, too)

• Force.com (Salesforce)
  – PaaS with integrated IaaS (public cloud)
  – (generalize SaaS app into a PaaS+IaaS)
Overview of Vendors

• **Microsoft Azure**
  – PaaS and IaaS provider (public cloud)
  – tries to follow Amazon

• **IBM**
  – PaaS + IaaS
  – separate PaaS and IaaS offerings, private cloud
  – high margin

• **Many, many Start-ups**
  – SaaS and PaaS on different clouds (e.g., 28msec)
  – IaaS (e.g., Rackspace, Eucalyptus)
  – many different kinds of business models
Amazon Web Services: IaaS++

- S3, CloudFront
  - storage of files
- EBS
  - virtual disk
- EC2
  - virtual machine
- SQS
  - queue
- SimpleDB, DynamoDB
  - indexing, simple queries
- DevPay, FPS
  - payment system
- Elastic MapReduce
  - Analytics
- Mechanical Turk, FWS
  - not IT related

Very dynamic: new services, changes all the time. All have REST APIs, but there are no standards!!!
History

• Starting point: Amazon is an online retailer
  – operates with high volume & low margins (like Migros)
  – operates globally
  – has a huge IT infrastructure
    • hardware, software, and talent
    • market power (special deals with HW vendors)
  – has a great payment system

• Starts new business line based on these assets
  – Step 1: make IT infrastructure available to the public
  – Step 2: help IT businesses make business
  – (big investments have already been made for retail)
• S3 = Simple Storage Service
• Reads+Writes object based on URI (REST, SOAP)
  – get(URI)
  – put(URI, bytestream)
• Data Model
  – Objects can be any bytestream
  – Objects can vary in size: 1 Byte – 5 Gbytes
  – Objects are associated to meta-data (4 KB)
  – Objects are organized in buckets
  – Possibility to scan all objects of a bucket
  – Security can be implemented for objects, buckets
S3

client

get(1/2) put(1/2, „ABC“) get(2) ...

Bucket 1

Bucket 2

...
S3 Details and Internals

• Very rich API that has evolved over time; e.g.,
  – `get-if-modified-since(uri, timestamp)`
  – `get-metadata(uri)`
  – ...

• Consistency, Availability, Partitioning
  – S3 focusses on „availability“ and „partitioning“
  – replicates objects; potentially, copies in multiple DC
  – propagates changes asynchronously (last update wins)
  – no guarantee that updates will be seen immediately called „eventual consistency“ in the literature
  – exact protocol and implementation not published!
S3 Pricing

• **Warning:** Subject to Change!!! (Status: Fall ’09)

• **Per Request**
  – USD 1 per 1MM „get“ requests
  – USD 10 per 1MM „update“ requests (POST, COPY, ...)

• **Data transfer**
  – In: USD 0.10 per 1GB
  – Out: ~USD 0.17 per 1GB (depending on volume)

• **Storage**
  – ~USD 0.15 per 1GB per month (depending on volume)
S3 vs. Disks (storage cost)

• **Disk**
  – USD 70 for 160 GB disk drive (Seagate, 2008)
  – replication factor of 2 (mirroring, RAID 2)
  – average life time: 2 years
  – ~USD 0.04 per 1GB per month (without electricity)

• **S3 (storage only!)**
  – USD 0.15 per 1GB per month

• **S3 wins if storage utilization is below ~25%**
  – what is the storage utilization of your disks?
S3 vs. Disk (performance)

- **Latency for Random Access (10 KB)**
  - S3 (ext): 140 ms
  - Disk: 10ms

- **Bandwidth for Sequential Access**
  - S3 (ext): 0.3 MB / sec
  - Disk: 100 MB / sec

- **Within AWS Datacenter (from EC2 machine)**
  - differences between S3 and local disk much smaller
  - (I do not have reliable numbers here)
Popular S3 Apps

• smugmug
  – photo sharing platform (like Flickr)
  – every photo stored as an object in S3
  – cache popular photos in smugmug farm (for cost)

• Backup (e.g., MySQL database)
  – store copy of whole DB or partition as S3 object

• General
  – data that is infrequently (or never) updated
  – data can be cached (cost) + consistency does not matter
  – lately, more aggressive use of S3 (-> 28msec PaaS on S3)
CloudFront

• “Variant“ of S3
  – integrates nicely with S3
  – same object model, use S3 API to „CRUD“ objects
• Designed for publishing documents
  – good if you have mostly „get“ request on objects
• Special feature: Global Replication
  – objects are replicated in different geographic regions
  – low latency + high bandwidth to read objects all around the world
• Pricing
  – USD 1 per 1MM get requests
  – ~USD 0.17 per 1GB traffic (depending on volume, region)
EC2

• Elastic compute cloud
• Rent a virtual machine (VM) by hour
  – Linux or Windows
  – load image of your app into VM
  – stop when you are done (possibly never)
• To persist data, need S3 (or EBS)
  – all data lost when you stop your VM or when it crashes
• How it works (nothing published)
  – if you use your VM, EC2 will find a real machine for you
  – if you do not use your VM, share machine with others
  – pay more when CPU is used
EC2 Pricing

• On demand, stand-by (low CPU utilization)
  – Small: USD 0.10 per hour
  – Large: USD 0.20 per hour
  – XL: USD 0.80 per hour
  – (no specs on machines published)

• Reserved machines, stand-by
  – Small: USD 227 per year or USD 350 for three years

• High CPU  2x more expensive than stand-by

• Windows ~25% more expensive than Linux

• Europe ~10% more expensive than USA

• Traffic from S3 to EC2 is free
EC2 vs. Server (cost)

• Hosted Server (with Internet Connectivity)
  – USD 1200 per year (e.g. Server4you)
  – clearly more expensive, independent of utilization

• Private Server (no Internet Connectivity)
  – USD 1000 for purchase
  – life time of two years
  – about same price if heavily used (100% utilization)
  – cheaper if high data requirements (e.g., bulkload)
Amazon EC2 Console Dashboard

Getting Started

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Launch Instances

Note: Your instances will launch in the US-East region.

Service Health

<table>
<thead>
<tr>
<th>Current Status</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon EC2 (US)</td>
<td>Loading...</td>
</tr>
</tbody>
</table>

My Resources

You are using the following Amazon EC2 resources in the US-East region:

- 3 Running Instances
- 2 EBS Volumes
- 13 Key Pairs
- 0 Elastic IPs
- 1 EBS Snapshot
- 5 Security Groups

Related Links

- Documentation
- All EC2 Resources
- Forums
- Feedback
- Report an Issue
EBS

- **EBS** = elastic block service
  - virtual disk
  - can be mounted from any EC2 instance
  - But, only one EC2 instance at a time (no concurrency)

- **Designed to support DB workloads**
  - EC2 instance runs database server
  - much faster scans

- **Pricing**
  - Requests: USD 0.01 for 1MM requests (read+write)
  - Storage: ~USD 0.15 per 1GB per month (dep. on vol.)
EBS vs. S3

• **EBS**
  – much faster and cheaper
  – access through a single EC2 instance
  – manual fail-over if EC2 instance fails

• **S3**
  – richer API and data model (e.g., metadata, TTL)
  – access from everywhere
  – 100% availability
Summary: IaaS

• Provide computing (HW) resources on demand
• Pioneered by Amazon (AWS)
• Works fairly well 😊
  – implements principles of cloud computing
  – delivers on (most) promises of cloud computing
  – many people + organizations do it already (Joe Doe – Gov.)
• Problems 😞
  – vendor lock-in: lack of standardization of APIs
  – SLAs: lack of standardization, auditing, enforcement
• In the long run, few real IaaS providers (commodity)
  – aka hardware industry: low margins -> large volume
  – difficult to run large data centers around the globe
Summary of AWS: Wish List

• Cost
  – utilization of hardware and software (ideal is ~100%)
  – pay-as-you-go & efficient (no overheads)
  – no vendor lock-in

• Time to market
  – focus on business problem (not IT)
  – no configuration, no provisioning, automatic security etc.
  – development framework (for enterprise Web apps)

• Operating & Support (-> cost + time-to-market)
  – SLAs: availability, guaranteed response times, ...
  – security (but might improve!)
  – elasticity: scale-out and down with workload
  – multi-tenancy (support for SaaS)
Why IaaS is not enough?

- IaaS a great starting point
- Unfortunately, only a fraction of the stack

![Layered diagram showing hardware, DBMS, application server, application, and customization, training, operation, ...]
Web Apps and the Cloud

• IaaS a great starting point
• Unfortunately, only a fraction of the stack

| Hardware | DBMS | Application Server | Application | Customization, Training, Operation, ... |
What does PaaS do?

- "System" Software Components aaS
  - Web server, App Server, Database, Queues, etc.
  - Security: auditing, encryption, ...

- Deployment of Apps via REST / WS APIs
  - often also development & debugging tools
  - data bulkloading tools

- Automation of Operating the Deployment
  - Fail-over: backup/recovery of data, activate VMs
  - Optimization, scalability, payment
Amazon Web Services: PaaS++

• S3, CloudFront
  – storage of files
• EBS
  – virtual disk
• EC2
  – virtual machine
• SQS
  – queue

• SimpleDB, DynamoDB
  – indexing, simple queries
• DevPay, FPS
  – payment system
• Elastic MapReduce
  – Analytics
• Mechanical Turk, FWS
  – not IT related
Summary of AWS: Wish List

• **Cost**
  – utilization of hardware and software (ideal is ~100%)
  – **pay-as-you-go** & efficient (no overheads)
  – **no vendor lock-in**

• **Time to market**
  – focus on business problem (not IT)
  – **no configuration, no provisioning, automatic security etc.**
  – **development framework (for enterprise Web apps)**

• **Operating & Support (-> cost + time-to-market)**
  – SLAs: availability, guaranteed response times, ...
  – **security** (but might improve!)
  – **elasticity: scale-out and down with workload**
  – multi-tenancy (support for SaaS)
Overview

• What and why cloud computing?
  – technologies and promises

• Some examples
  – Amazon Web Services

• **Databases in the Cloud**
  – transactions (many small reads & updates – OLTP)
Reference Architecture for DB Apps

- **Client**
  - HTTP
  - XML, JSON, HTML

- **Web Server**
  - FCGI, ...
  - XML, JSON, HTML

- **App Server**
  - SQL
  - records

- **DB Server**
  - get/put
  - block

- **Store**
Open Questions

- How to map stack to IaaS?
- How to implement store layer?
- What consistency model?
- What programming model?
- Whether and how to cache?
Open Questions

• **How to map stack to IaaS?**
  • How to provision resources?

• How to implement store layer?

• What consistency model?

• What programming model?

• Whether and how to cache?
Variant I: Partition Workload by „Tenant“

- Client
  - HTTP
  - XML, JSON, HTML
  - Web Server
  - FCGI, ...
  - XML, JSON, HTML
  - App Server
  - SQL
  - records
  - DB Server
  - get/put
  - block
  - Store

- Workload Splitter
  - XML, JSON, HTML
  - Server-A
  - Server-B
  - Server-A
  - Server-B
  - Store-A
  - Store-B
Partition Workload by „Tenant“

• **Principle**
  – partition data by „tenant“
  – route request to DB of that tenant

• **Advantages**
  – reuse existing database stack (RDBMS)
  – flexibility to use DAS or SAN/NAS

• **Disadvantages**
  – multi-tenant problem [*Salesforce*]
    • optimization, migration, load balancing, fix cost
  – silos: need DB federator for inter-tenant requests
  – expensive HW and SW for high availability
If a shop is successful, you need to move it!

(popularity vs. growth of product assortment)
Summary of Variant I: Wish List

• Cost
  – utilization of hardware and software (ideal is ~100%)
  – pay-as-you-go & efficient (no overheads)
  – no vendor lock-in

• Time to market
  – focus on business problem (not IT)
  – no configuration, no provisioning, automatic security etc.
  – development framework (for enterprise Web apps)

• Operating & Support (-> cost + time-to-market)
  – SLAs: availability, guaranteed response times, ...
  – security
  – elasticity: scale-out and down with workload
  – multi-tenancy (support for SaaS)
Variant II: Partition Workload by „Request“

- **Client**
  - HTTP
  - Web Server
    - FCGI, ...
  - App Server
    - SQL
    - get/put block
  - DB Server
    - records
  - Store
- **Store** (e.g., S3)
- **Workload Splitter**
  - XML, JSON, HTML
  - Server-A
  - Server-B
  - ???

Client

Client

Client
Metaphor: Internet Department Store

- If a product is successful, you stock up its supply
  - Transparent and fine-grained reprovisioning
  - Cost of reprovisioning much lower!!!
Partition Workload by „Request“

• **Principle**
  – fine-grained data partitioning by page or object
  – any server can handle any request, stateless servers
  – implement DBMS as a library (not server)

• **Advantages**
  – avoids disadvantages of Variant I

• **Disadvantages**
  – new synchronization problem (→ CAP theorem)
  – only works with separate storage tier (no DAS!)
  – whole new breed of systems
  – caching not effective
Summary of Variant II: Wish List

• Cost
  – utilization of hardware and software (ideal is ~100%)
  – pay-as-you-go & efficient (no overheads)
  – no vendor lock-in

• Time to market
  – focus on business problem (not IT)
  – no configuration, no provisioning, automatic security etc.
  – development framework (for enterprise Web apps)

• Operating & Support (-> cost + time-to-market)
  – SLAs: availability, guaranteed response times, ...
  – security
  – elasticity: scale-out and down with workload
  – multi-tenancy (support for SaaS)
Variant I vs. Variant II

• **Simplification of architecture great**
  – performance: avoid unnecessary duplicate work
  – cost: save cost for licenses of unnecessary layers
  – flexibility: evolve app easier because one model

• **Simplification of workload splitter good**
  – Route requests round-robin vs. „tenant“
  – No migration or special optimization needed
    no danger of getting that wrong

• **Maturity of products**
  – No Variant II product is mainstream today
  – (but, IT forces of cloud, Web, and new HW might be strong enough for a major shift)
Overview of DaaS Vendors

- **Microsoft Azure, Amazon RDS – Camp 1**
  - database + complete systems stack
  - development & debugging & deployment tools

- **Google AppEngine – Camp 2**
  - database + complete systems stack
  - development & debugging & deployment tools

- **Many, Start-ups – both Camps**
  - variety of tools and models
Evaluating the State of the Art
Promises of Cloud Computing

• **Cost**
  – reduce cost
    • utilization, commoditization, prevention + having failures
    • better optimization: statistics, experts, transparent upgrades
  – “pay as you go“ (cap-ex vs. op-ex)
  – easy to test!

• **Time to market**
  – avoid unnecessary steps
    • HW provisioning, purchasing, testing
  – difficult to test!
    • can only test scalability / elasticity
What to optimize and measure?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Traditional</th>
<th>Cloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost [$]</td>
<td>fixed</td>
<td>optimize</td>
</tr>
<tr>
<td>Performance [tps, secs]</td>
<td>optimize</td>
<td>fixed</td>
</tr>
<tr>
<td>Scale-out [#cores]</td>
<td>optimize</td>
<td>fixed</td>
</tr>
<tr>
<td>Predictability [$\sigma($)]</td>
<td>-</td>
<td>fixed</td>
</tr>
<tr>
<td>Consistency [%]</td>
<td>fixed</td>
<td>???</td>
</tr>
<tr>
<td>Flexibility [#variants]</td>
<td>-</td>
<td>optimize</td>
</tr>
</tbody>
</table>
Benchmarking Cloud Services

• Goal: Test if cloud promises are fulfilled

• Benchmark workload [DBTest09, SIGMOD10]
  – TPC-W Benchmark, Ordering Mix
  – Adapted for testing elasticity and cost ($)
  – Vary load (EB): clicks per second

• Benchmark metrics
  – Cost: $ / WI
  – Cost Predictability: $\sigma(\$/WI)$
  – Throughput: WIPS
## Tested Services

<table>
<thead>
<tr>
<th>Tested Services</th>
<th>MS Azure</th>
<th>Google App Eng</th>
<th>AWS RDS</th>
<th>28msec (S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Model</td>
<td>PaaS</td>
<td>PaaS</td>
<td>PaaS</td>
<td>IaaS</td>
</tr>
<tr>
<td>Architecture</td>
<td>Replication</td>
<td>Part. + Repl. (+Dist. Control)</td>
<td>Classic</td>
<td>Distr. Control</td>
</tr>
<tr>
<td>Consistency</td>
<td>SI</td>
<td>≈ SI</td>
<td>Rep. Read</td>
<td>EC</td>
</tr>
<tr>
<td>Cloud Provider</td>
<td>Microsoft</td>
<td>Google</td>
<td>Amazon</td>
<td>Flexible</td>
</tr>
<tr>
<td>Web/App Server</td>
<td>.Net Azure</td>
<td>AppEngine</td>
<td>Tomcat</td>
<td>Tomcat</td>
</tr>
<tr>
<td>Database</td>
<td>SQL Azure</td>
<td>DataStore</td>
<td>MySQL</td>
<td>--</td>
</tr>
<tr>
<td>Storage / FS</td>
<td>Simple DataStore</td>
<td>GFS</td>
<td>--</td>
<td>S3</td>
</tr>
<tr>
<td>App-Language</td>
<td>C#</td>
<td>Java/AppEngine</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td>DB-Language</td>
<td>SQL</td>
<td>GQL</td>
<td>SQL</td>
<td>Low-Lev. API</td>
</tr>
</tbody>
</table>
## Tested Services

<table>
<thead>
<tr>
<th></th>
<th>AWS SimpleDB</th>
<th>AWS MySQL</th>
<th>AWS MySQL/R</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Business Model</strong></td>
<td>PaaS</td>
<td>IaaS</td>
<td>IaaS</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Replication</td>
<td>Classic</td>
<td>Replication</td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
<td>EC</td>
<td>Rep. Read</td>
<td>Rep. Read</td>
</tr>
<tr>
<td><strong>Cloud Provider</strong></td>
<td>Amazon</td>
<td>Flexible</td>
<td>Flexible</td>
</tr>
<tr>
<td><strong>Web/App Server</strong></td>
<td>Tomcat</td>
<td>Tomcat</td>
<td>Tomcat</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>SimpleDB</td>
<td>MySQL</td>
<td>MySQL</td>
</tr>
<tr>
<td><strong>Storage / File System</strong></td>
<td>--</td>
<td>EBS</td>
<td>EBS</td>
</tr>
<tr>
<td><strong>App-Language</strong></td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td><strong>DB-Language</strong></td>
<td>SimpleDB Queries</td>
<td>SQL</td>
<td>SQL</td>
</tr>
<tr>
<td></td>
<td>EBs</td>
<td>Fully Utilized</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>MySQL</td>
<td>0.635</td>
<td>0.072</td>
<td>0.020</td>
</tr>
<tr>
<td>MySQL/R</td>
<td>2.334</td>
<td>0.238</td>
<td>0.034</td>
</tr>
<tr>
<td>RDS</td>
<td>1.211</td>
<td>0.126</td>
<td>0.032</td>
</tr>
<tr>
<td>SimpleDB</td>
<td>0.384</td>
<td>0.073</td>
<td>0.042</td>
</tr>
<tr>
<td>S3</td>
<td>1.304</td>
<td>0.206</td>
<td>0.042</td>
</tr>
<tr>
<td>Google AE</td>
<td>0.002</td>
<td>0.028</td>
<td>0.033</td>
</tr>
<tr>
<td>Google AE/C</td>
<td>0.002</td>
<td>0.018</td>
<td>0.026</td>
</tr>
<tr>
<td>Azure</td>
<td>0.775</td>
<td>0.084</td>
<td>0.023</td>
</tr>
</tbody>
</table>
Experiments: Cost / WI (m$)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Low Load</th>
<th>Peak Load</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant 1 (RDS)</td>
<td>1.212</td>
<td>0.005</td>
<td>0.154</td>
</tr>
<tr>
<td>Variant 2 (S3, 28ms)</td>
<td>-</td>
<td>0.007</td>
<td>-</td>
</tr>
<tr>
<td>Variant 2 (Google)</td>
<td>0.002</td>
<td>0.028</td>
<td>0.011</td>
</tr>
<tr>
<td>Variant 1 (Azure)</td>
<td>0.775</td>
<td>0.005</td>
<td>0.058</td>
</tr>
</tbody>
</table>

Only V2 is cheap and predictable!
Cost predictability [m$/WI]

<table>
<thead>
<tr>
<th>Service</th>
<th>mean ± σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>0.015 ± 0.077</td>
</tr>
<tr>
<td>MySQL/R</td>
<td>0.043 ± 0.284</td>
</tr>
<tr>
<td>RDS</td>
<td>0.030 ± 0.154</td>
</tr>
<tr>
<td>SimpleDB</td>
<td>0.063 ± 0.089</td>
</tr>
<tr>
<td>S3</td>
<td>0.018 ± 0.098</td>
</tr>
<tr>
<td>Google AE</td>
<td>0.029 ± 0.016</td>
</tr>
<tr>
<td>Google AE/C</td>
<td>0.021 ± 0.011</td>
</tr>
<tr>
<td>MS Azure</td>
<td>0.010 ± 0.058</td>
</tr>
</tbody>
</table>

- **Ideal:** Cost / WI is constant
  - low σ better (ideal: σ=0)
  - cost independent of load
- **Google clear winner here!**
## Cost per Day [$], Vary EB

<table>
<thead>
<tr>
<th>Service</th>
<th>1</th>
<th>250</th>
<th>1000</th>
<th>3000</th>
<th>9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>8.25</td>
<td><strong>22.85</strong></td>
<td>64.48</td>
<td>183.66</td>
<td>-</td>
</tr>
<tr>
<td>MySQL/R</td>
<td>16.40</td>
<td>31.01</td>
<td>72.90</td>
<td>194.44</td>
<td>-</td>
</tr>
<tr>
<td>RDS</td>
<td>14.75</td>
<td>29.47</td>
<td>70.92</td>
<td>187.88</td>
<td></td>
</tr>
<tr>
<td>SimpleDB</td>
<td>4.42</td>
<td>116.52</td>
<td>412.25</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>S3</td>
<td>16.02</td>
<td>71.96</td>
<td>131.78</td>
<td>346.60</td>
<td>914.87</td>
</tr>
<tr>
<td>Google AE</td>
<td><strong>0.02</strong></td>
<td>82.25</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Google AE/C</td>
<td><strong>0.02</strong></td>
<td>60.57</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Azure</td>
<td>9.15</td>
<td>22.86</td>
<td><strong>61.22</strong></td>
<td>176.87</td>
<td>521.13</td>
</tr>
</tbody>
</table>
Cost Factors 250EBs [\$]

![Diagram showing cost factors for various services including MySQL, RDS, SDB, S3, AE/C, and Azure. The chart demonstrates the cost per day in dollars, with different colored bars representing network, fixed CPU, variable CPU, and storage costs.](image_url)
Relative Cost Factors

MySQL

Azure

SimpleDB

S3

Network

Fixed CPU

Variable CPU

Storage

Network

Fixed CPU

Network

Network

Network

Red

Orange

Yellow

Blue
High performance can be achieved with both!

Warning: Still a hypothesis. S3 is only EC.
Overload Behavior

RDS

SimpleDB

App Engine

WIPS Issued
WIPS in RT
Ideal

Emulated Browsers

Emulated Browsers

Emulated Browsers
Summary

• Promises of Cloud Computing
  – reduce cost, reduce time-to-market

• Principles and Background of Cloud Computing
  – Automation, Virtualization, Pay-as-you-go

• Kinds of Clouds
  – Public, private, hybrid, Cloud on Chip, swarms, ...
  – IaaS vs. PaaS vs. SaaS

• Distributed & Parallel Database Systems
  – Query Processing
  – Transaction Processing
Summary

• There are many offerings
  – big players (Amazon, Google, Microsoft)
  – start-ups (e.g., 28msec and many more)

• Plenty of homework: Still wild west!!!
  – Efficient implementation: reduce differences
  – Standardization: NIST is working on this

• There are fundamentally different approaches
  – tradeoffs in cost, scalability, consistency, security, ...
  – no clear winner so far; might depend on requirements
  – possible separation of „enterprise“ vs. „Internet“
Backup Slides

• further reading (if you are interested :-) )
• not required for exam
Web Services: Standards

- **URI**
  - a way to give a name to a service

- **SOAP**
  - data format (XML for envelop and body)
  - specify bindings to protocols and intermediaries

- **WSDL**
  - define interface, message types, supported bindings

- **UDDI**
  - white, yellow, green pages of services (directory)

- **Or simply use REST**
  - URI + standardized http/https interface (get, post, ...)
Application Integration
Application Integration
What happens in case of a delay?
What happens if I change an interface?
How can I optimize this process?
What about people?
Common Issues

• Platform dependency
• Management and Optimization
  – What does 99.99% availability mean?
  – How do I guarantee 3 seconds response time?
  – Who owns the log?
  – Load balancing, caching, replication?
• Change of process: all or nothing
  – failure or upgrade to new version
  – change message format during long running workflow
Loose Coupling of Applications
Lose Kopplung von Anwendungen (CORBA)

Good old CORBA!!!
Lose Kopplung von Anwendungen (CORBA)

Good old CORBA!!!
Lose Kopplung von Anwendungen (Web Services)

Web Services
Virtualisation of Applications

Web Services

I want all x!

Find & Bind
Virtualisation of Applications

Web Services
Virtualisierung of Applications

Web Services

I want all $x$!

$x_1$

$WSDL$

$App$

Message Broker

$x_2, x_3$

Find & Bind

$I want y!$

$I want y!$
Virtualisation of Applications

Web Services
Open Questions

• How to map traditional DB stack to IaaS?
• **How to implement the storage layer?**
• What is the right consistency model?
• What is the right programming model?
• Whether and how to make use of caching?
Storage Layer Questions

- **What is the interface?**
  - get/put or more?
  - return records or blocks?

- **How to implement this interface?**
  - data structures
  - storage media and network
  - clustering of data
  - granularity of partitioning / sharding
API of Store

• Assumption: Hardware trends will change the game
  – most data in MM or SSD / PCM

• Proposition 1: return tuples rather than blocks
  – blocks are an arte-fact of old HW (i.e., disks)

• Proposition 2: push down predicates
  – because it can be done (power depends on impl.)
  – compensates for cost of remote storage
  – (old idea; e.g., iDisks)

• Open: consistency construct of store?
  – store might have nice properties (e.g., mon. Writes)
  – store might have nice primitives (e.g., counter, set/test)
Store Implementation Variants

• **DAS**
  – local disks with physically exclusive access
  – put/get interface; no synchronization
  – only works for V1

• **Key-value stores (e.g., Dynamo)**
  – DHTs with concurrent access
  – put/get interface; no synchronization
  – works for V1 and V2; makes more sense for V2

• **A DBMS (e.g., MySQL, SQL Server)**
  – have been misused before

• **ClockScan [Unterbrunner et al. 2009]**
  – massively shared scans in a distributed system
Open Questions

• How to map traditional DB stack to IaaS?
• How to implement the storage layer?
• **What is the right consistency model?**
• What is the right programming model?
• Whether and how to make use of caching?
CAP Theorem

• Three properties of distributed systems
  – Consistency (ACID transactions w. serializability)
  – Availability (nobody is ever blocked)
  – resilience to network Partitioning

• Result
  – it is trivial to achieve 2 out of 3
  – it is impossible to have all three

• Two schools
  – Databases: sacrifice availability
  – Distributed systems: sacrifice consistency
Why and where is CAP a problem?

• Everybody wants to have it all
  – C: hard on programmers to give up
  – A: hard on business to give up
  – P: impossible to avoid (fact of life)

• CAP: Across data centers
  – All people give up „C“ (in some form), even if A violated
  – Reason: latency

• CAP: Within a data center in V2
  – Some people give up „C“ (somehow), even if P violated
  – Reason: cost

• Most of the discussion is on „C“  [Vogels, VLDB 2007]
What have people done?

• **Client-side Consistency Models** [Tannenbaum]
  – read & write monotonicity, session consistency, ...

• **Time-line consistency** [PNUTS08]
  – all writes are consistent (except across DC)
  – read consistency: either don’t care or latest version

• **New DB transaction models**
  – Escrow, Reservation Pattern [O’Neil 86], [Gawlick 09]
  – SAGAs and compensation; e.g., in BPEL [G.-Molina,Salem]
  – SAP, Amadeus et al. [Buck-Emden], [Kemper et al. 98]

• **Define logical unit of consistency** [Azure]
  – strong consistency within LuC
  – app-defined consistency across LuCs

• **Educate Application Developers** [Helland 2009]
What have people done?

• **Client-side Consistency Models** [Tannenbaum]
  – read & write monotonicity, session consistency, ...

• **Time-line consistency** [PNUTS08]
  – all writes are consistent (except across DC)

• **New DB transaction models**
  – Escrow, Reservation Pattern [O'Neil 86], [Gawlick 09]
  – SAGAs and compensation; e.g., in BPEL [G.-Molina, Salem]
  – SAP, Amadeus et al. [Buck-Emden], [Kemper et al. 98]

• **Define logical unit of consistency** [Azure]
  – strong consistency within LuC
  – app-defined consistency across LuCs

• **Educate Application Developers** [Helland 2009]

  Push problem to app programmer!
What have we done?

• **Levels of Consistency, Cost Tradeoffs** [Brantner08]
  – can achieve read/write monotonicy + „A“ + „P“
  – the more consistency, the higher the cost

• **Economic models for consistency** [Amadeus], [Kraska09]
  – Classify the data as A, B, C
    • A data always strong consistent
    • C data always eventually consistent
    • B data handled like A or C data adaptively
  – Cost model that estimates business impact of inconst.
What have we done?

• Levels of Consistency, Cost Tradeoffs [Brantner08]
  – can achieve read/write monotonicy + „A“ + „P“
  – Cost model that estimates business impact of inconst.

• Economic models for consistency [Amadeus], [Kraska09]
  – B data handled like A or C data adaptively
  – Follow tradition to push problem to app programmer!
Cost per 1000 TAs [\$] (TPC-W)

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Adjustable</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing</td>
<td>0.15</td>
<td>0</td>
<td>0.15</td>
</tr>
<tr>
<td>Durability</td>
<td>1.8</td>
<td>1.1</td>
<td>0.7</td>
</tr>
<tr>
<td>Monotonicity</td>
<td>2.1</td>
<td>1.4</td>
<td>0.7</td>
</tr>
<tr>
<td>Mono+Atomicity</td>
<td>2.9</td>
<td>2.6</td>
<td>0.3</td>
</tr>
</tbody>
</table>

[Brantner+08]
### Latency per TAs [secs] (TPC-W)

<table>
<thead>
<tr>
<th></th>
<th>Avg.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing</td>
<td>11.3</td>
<td>12.1</td>
</tr>
<tr>
<td>Durability</td>
<td>4.0</td>
<td>5.9</td>
</tr>
<tr>
<td>Monotonicity</td>
<td>4.0</td>
<td>6.8</td>
</tr>
<tr>
<td>Mono+Atomicity</td>
<td>2.8</td>
<td>4.6</td>
</tr>
</tbody>
</table>

*[Brantner+08]*
Open Questions

- How to map traditional DB stack to IaaS?
- How to implement the storage layer?
- What is the right consistency model?
- What is the right programming model?
- Whether and how to make use of caching?
Programming Model

- Properties of a programming lang. for the cloud
  - support DB-style + OO-style + CEP-style
  - avoid keeping state at app servers for V2
- Many languages will work in the cloud
  - SQL, XQuery, Ruby, .Net /LINQ, ...;
  - J2EE will not work
- Open (research) questions
  - do OLAP on the OLTP data: My guess is yes!
  - rewrite your apps: My guess is yes because of CAP!
Caching

- Many Variants Possible
  - this is just one
  - V1 caching mandatory
  - V2 caching prohibitive

- TPC-W Experiments
  - marginal improvements for Google AppEngine

- No low hanging fruit
Other AWS Services

- SQS
- SimpleDB (now DynamoDB)
- RDS
- Auto Scale
- DevPay
- Amazon vs. Google vs. Microsoft vs. others
PaaS by Amazon

• **Data Management Services**
  – SQS: Queues (for workflow management)
  – SimpleDB: Simple Database (for Web 2.0)
  – RDS: SQL Database (for OLTP)

• **Support for operations**
  – AutoScale: scalability
  – DevPay: billing
SQS

• SQS = Simple Queuing Service
  – `createQueue(uri)`
  – `grant(uri, user)`
  – `send(uri, msg)`
  – `receive(uri, number-msg, timeout)`
  – `delete(uri, msg-id)`

• Reading from a queue
  – `receive` implicitly sets lock so msgs not processed twice
  – lock has timeout so msg not lost if reader fails

• SQS gives no FIFO completeness guarantees
  – eventual consistency spirit like S3
SQS Pricing and Performance

• **Pricing**
  – send: USD 10 per 1MM msgs
  – Data transfer: USD 0.17 per GB (same as S3)

• **Latency (ext. client, round-trip with ack)**
  – send: 0.31 secs
  – receive: 0.16 secs
  – delete: 0.16 secs
  – independent of size of message
SimpleDB

- **Simple = Simple Database**
  - `createDomain`, `deleteDomain`, `domainMetadata`
  - `put`, `batchPut`, `delete`
  - `getAttributes`, `select`

- **Data model**
  - Semi-structured (~Microsoft Sharepoint)

- **Use cases**
  - As a database or as an index to S3 objects
SimpleDB Pricing & Performance

• **Cost**
  – USD 0.14 per machine hour (query processing)
  – USD 0.10 per GB of data transfer

• **Performance**
  – highly dependent on query
  – for simple queries, like S3
RDS

• **Technically**
  – run a MySQL server on EC2 and EBS

• **Services**
  – launch/shut-down DB Instance (MySQL server)
  – automatic fail-over if EC2 instance fails

• **Pricing**
  – Derived from EC2: data transfer + CPU
  – (depends on size of CPU, location of data center)
Auto Scaling

• **Add-on to EC2**
  – monitors load of EC2 instances
  – Shrinks / grows #EC2 depending on user-def. Rules

• **Simplifies operating significantly**
  – Solves scalability issues at „app server“ level
  – Helps with failure of EC2 machines

• **Pricing**
  – Free (but, of course, need to pay for EC2)
DevPay

• Opens door for AWS-value added resellers
  – E.g., SaaS and PaaS vendors

• Principle: Revenue Sharing
  – Vendor offers a service on AWS (e.g., car pooling)
  – Users use that service on their AWS account
  – Users pay a mark-up on AWS consumption

• Example: Car pooling
  – User pays for USD 1.00 per EC2 hour for app
  – Amazon takes USD 0.10 per EC2 hour
  – Provider gets USD 0.90 per EC2 hour

• FPS: Allows other business models on top of AWS
Amazon Bill

• Lists cost for all services
  – IaaS: S3, EC2, EBS, ...
  – PaaS: SQS, SimpleDB, RDS, ...
  – mark-up of third-party vendors (DevPay)

• Possibility to view details
  – XML of all requests, starts of VMs, etc.
<table>
<thead>
<tr>
<th>Service</th>
<th>Usage</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Elastic Compute Cloud</td>
<td></td>
<td>129.24</td>
</tr>
<tr>
<td>Amazon Elastic MapReduce</td>
<td></td>
<td>0.00</td>
</tr>
<tr>
<td>Amazon Simple Queue Service</td>
<td></td>
<td>0.00</td>
</tr>
<tr>
<td>Amazon Simple Storage Service</td>
<td></td>
<td>1.27</td>
</tr>
<tr>
<td>Amazon SimpleDB</td>
<td></td>
<td>0.10</td>
</tr>
<tr>
<td><strong>Taxes</strong></td>
<td></td>
<td>0.00</td>
</tr>
<tr>
<td><strong>Charges due on September 1, 2009</strong></td>
<td></td>
<td><strong>130.61</strong></td>
</tr>
</tbody>
</table>

All web services are sold by Amazon Web Services LLC.
How to build Apps on AWS: Small

- **Client**
- **Web Server**
- **App Server**
- **DB Server**
- **Store**

**HTTP**

**XML, JSON, HTML**

**FCGI, ...**

**XML, JSON, HTML**

**SQL**

**records**

**get/put**

**block**

**EC2**

**EC2**

**RDS**

**EBS**
How to build Apps on AWS: Large
Bets Made by AWS

• How to map traditional DB stack to IaaS?
  – implemented V1 (SimpleDB, RDS) and V2 (SQS)
  – provide IaaS so that start-ups can use both

• How to implement the storage layer?
  – SAN for EBS, key-value store (Dynamo) for S3
  – MySQL for RDS

• What is the right consistency model?
  – Allow everything, support for EC and strong consistency

• What is the right data + programming model?
  – Support everything; support for embedded SQL

• Whether and how to make use of caching?
  – No! (leave it to the app developer, no specific support)
Google AppEngine

• **Complete Systems Stack**
  – Applications: Java or Python with embedded SQL
  – SQL: dramatic simplification (no joins, group bys)
  – Java: dramatic simplification (no session beans)

• **API**
  – Deploy application
  – Manage versions of application

• **Pricing**
  – Pay per CPU hour (like EC2) + traffic; some free CPU hrs
  – Warning: how do you know that bill is correct?
## AWS vs. Google AE (cost)

<table>
<thead>
<tr>
<th></th>
<th>AWS</th>
<th>Google AE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (EC2)</td>
<td>USD 0.10 / 0.20 per hour</td>
<td>USD 0.10 per hour</td>
</tr>
<tr>
<td>Data Transfer (S3)</td>
<td>~ USD 0.10 / GB</td>
<td>~USD 0.10 / GB</td>
</tr>
<tr>
<td>Stored Data (S3)</td>
<td>~ USD 0.15 / GB x month</td>
<td>USD 0.15 / GB x month</td>
</tr>
</tbody>
</table>

- Is this a coincidence? Not a healthy market yet!
- Not an apples to apples comparison
  - Google AE is PaaS with integrated IaaS
  - difficult to estimate compute hours in AE
    - it took us 10 minutes to consume 36 hours (!)
Bets Made by Google

• How to map traditional DB stack to IaaS?
  – Variant II

• How to implement the storage layer?
  – KVS, BigTable

• What is the right consistency model?
  – Strong consistency

• What is the right data + programming model?
  – Python / Java + GQL

• Whether and how to make use of caching?
  – Yes! (data from database)
Microsoft Azure

• **Launched in February 2010**
  – Windows Azure: similar to EC2, but with add-ons
    • features: Web Server, .Net runtime, ...
  – SQL Azure: similar to RDS, but with restrictions (s.n.)

• **Constantly adding new services**
  – Will become similar to AWS over time

• **Pricing**
  – Windows Azure: similar to EC2 (CPU hrs + traffic)
  – SQL Azure: monthly flat fee depending on DB size
SQL Azure

- **Concept: Logical unit of consistency**
  - Can deploy DBs in several sizes: max. 64 GB
  - Guarantees consistency within on DB (i.e., LUC)
  - Appl. needs to federate between LUCs

- **Why does Azure do that?**
  - CAP theorem on consistency
  - Control cost of moving DBs in the cloud
How to build Apps on Azure

Client

Web Server

HTTP

XML, JSON, HTML

FCGI, ...

XML, JSON, HTML

App Server

Get/put block

DB Server

SQL

Records

SQL Azure

W.Azure-A

W.Azure-B

Client

Client

Client
Force.com / Salesforce

• **PaaS: Similar Architecture to Azure**
  – But uses proprietary programming language
  – Uses Oracle instead of SQL Server as DBMS

• **SaaS: Based on PaaS infrastructure**
  – Hosted CRM application for SME
  – Historically SaaS came before PaaS at Salesforce
  – Notion of tenant: one customer of SaaS
    • Packs many tenant into single Oracle database
    • Similar to EC2 which packs many VMs into single machine
    • But, MUCH more complicated (query optimization, etc.)
Force.com Architecture

**Client**
- HTTP
- XML, JSON, HTML
- Workload Splitter
- Server-A
- Server-B
- Store-A
- Store-B
- App Server
- FCGI, ...
- SQL
- get/put block
- Store
- Web Server
Bets Made by Microsoft + SF

• How to map traditional DB stack to IaaS?
  – Variant I

• How to implement the storage layer?
  – Relational database system (with local disk)

• What is the right consistency model?
  – Strong consistency (ISO SQL)

• What is the right data + programming model?
  – Proprietary languages

• Whether and how to make use of caching?
  – No!
28msec

- **Integrated Database, Application, and Web Server**
  - Proprietary database system, can also use RDBMS
  - App + Web Server: Tomcat
- **Programming Language**
  - XQuery; semi-structured data model (JSON, XML,...)
- **Runs in any cloud (public + private)**
- **Adopted both architecture variants**
  - Variant I: strong consistency, no fault tolerance
  - Variant II: consistency à la carte, fault tolerance
28msec (V2) on AWS

Client -> HTTP -> Web Server -> FCGI, ... -> App Server -> SQL -> DB Server -> get/put block -> Store

Client -> HTTP -> XML, JSON, HTML -> Workload Splitter -> EC2-A -> S3, SQS -> EC2-B
28msec Demo

- Development of Apps
- Deployment of Apps
- Bulkloading of Seed Data
- Versioning of Apps

- http://sausalito.28msec.com
Bets Made by 28msec

- **How to map traditional DB stack to IaaS?**
  - implemented both architectures (V1 + V2)
  - V1 only in a single server variant for low end

- **How to implement the storage layer?**
  - EBS and PostGres for V1; KVS for V2

- **What is the right consistency model?**
  - ACID for V1; configurable for V2

- **What is the right data + programming model?**
  - XML & XQuery

- **Whether and how to make use of caching?**
  - No! (Only for code / precompiled query plans)