Cloud Architectures

June 2008

Jinesh Varia
Technology Evangelist
Amazon Web Services
(jvaria@amazon.com)

Introduction

This paper illustrates the style of building applications using services available in the Internet cloud.

Cloud Architectures are designs of software applications that use Internet-accessible on-demand services. Applications built on Cloud Architectures are such that the underlying computing infrastructure is used only when it is needed (for example to process a user request), draw the necessary resources on-demand (like compute servers or storage), perform a specific job, then relinquish the unneeded resources and often dispose themselves after the job is done. While in operation the application scales up or down elastically based on resource needs.

This paper is divided into two sections. In the first section, we describe an example of an application that is currently in production using the on-demand infrastructure provided by Amazon Web Services. This application allows a developer to do pattern-matching across millions of web documents. The application brings up hundreds of virtual servers on-demand, runs a parallel computation on them using an open source distributed processing framework called *Hadoop*, then shuts down all the virtual servers releasing all its resources back to the cloud—all with low programming effort and at a very reasonable cost for the caller.

In the second section, we discuss some best practices for using each Amazon Web Service - Amazon S3, Amazon SQS, Amazon SimpleDB and Amazon EC2 - to build an industrial-strength scalable application.

Keywords

Amazon Web Services, Amazon S3, Amazon EC2, Amazon SimpleDB, Amazon SQS, Hadoop, MapReduce, Cloud Computing

Why Cloud Architectures?

Cloud Architectures address key difficulties surrounding large-scale data processing. In traditional data processing it is difficult to get as many machines as an application needs. Second, it is difficult to get the machines when one needs them. Third, it is difficult to distribute and co-ordinate a large-scale job on different machines, run processes on them, and provision another machine to recover if one machine fails. Fourth, it is difficult to auto-scale up and down based on dynamic workloads. Fifth, it is difficult to get rid of all those machines when the job is done. Cloud Architectures solve such difficulties.

Applications built on Cloud Architectures run in-the-cloud where the physical location of the infrastructure is determined by the provider. They take advantage of simple APIs of Internet-accessible services that scale on-demand, that are industrial-strength, where the complex reliability and scalability logic of the underlying services remains implemented and hidden inside-the-cloud. The usage of resources in Cloud Architectures is as needed, sometimes ephemeral or seasonal, thereby providing the highest utilization and optimum bang for the buck.

Business Benefits of Cloud Architectures

There are some clear business benefits to building applications using Cloud Architectures. A few of these are listed here:

1. *Almost zero upfront infrastructure investment*: If you have to build a large-scale system it may cost a fortune to invest in real estate, hardware (racks, machines, routers, backup power supplies), hardware management (power management, cooling), and operations personnel. Because of the upfront costs, it would typically need several rounds of management approvals before the project could even get started. Now, with utility-style computing, there is no fixed cost or startup cost.
2. *Just-in-time Infrastructure*: In the past, if you got famous and your systems or your infrastructure did not scale you became a victim of your own success. Conversely, if you invested heavily and did not get famous, you became a victim of your failure. By deploying applications in-the-cloud with dynamic capacity management software architects do not have to worry about pre-procuring capacity for large-scale systems. The solutions are low risk because you scale only as you *grow*. Cloud Architectures can relinquish infrastructure as quickly as you got them in the first place (in minutes).
3. *More efficient resource utilization:* System administrators usually worry about hardware procuring (when they run out of capacity) and better infrastructure utilization (when they have excess and idle capacity). With Cloud Architectures they can manage resources more effectively and efficiently by having the applications request and relinquish resources only what they need (on-demand).
4. *Usage-based costing*: Utility-style pricing allows billing the customer only for the infrastructure that has been used. The customer is not liable for the entire infrastructure that may be in place. This is a subtle difference between desktop applications and web applications. A desktop application or a traditional client-server application runs on customer's own infrastructure (PC or server), whereas in a Cloud Architectures application, the customer uses a third party infrastructure and gets billed only for the fraction of it that was used.
5. *Potential for shrinking the processing time*: Parallelization is the one of the great ways to speed up processing. If one compute-intensive or data-intensive job that can be run in parallel takes 500 hours to process on one machine, with Cloud Architectures, it would be possible to spawn and launch 500 instances and process the same job in 1 hour. Having available an elastic infrastructure provides the application with the ability to exploit parallelization in a cost-effective manner reducing the total processing time.

Examples of Cloud Architectures

There are plenty of examples of applications that could utilize the power of Cloud Architectures. These range from back-office bulk processing systems to web applications. Some are listed below:

- Processing Pipelines
  - Document processing pipelines – convert hundreds of thousands of documents from Microsoft Word to PDF, OCR millions of pages/images into raw searchable text
  - Image processing pipelines – create thumbnails or low resolution variants of an image, resize millions of images
  - Video transcoding pipelines – transcode AVI to MPEG movies
  - Indexing – create an index of web crawl data
  - Data mining – perform search over millions of records
- Batch Processing Systems
  - Back-office applications (in financial, insurance or retail sectors)
  - Log analysis – analyze and generate daily/weekly reports
  - Nightly builds – perform nightly automated builds of source code repository every night in parallel
  - Automated Unit Testing and Deployment Testing – Test and deploy and perform automated unit testing (functional, load, quality) on different deployment configurations every night
- Websites
  - Websites that "sleep" at night and auto-scale during the day
  - Instant Websites – websites for conferences or events (Super Bowl, sports tournaments)
  - Promotion websites
  - "Seasonal Websites" - websites that only run during the tax season or the holiday season ("Black Friday" or Christmas)

In this paper, we will discuss one application example in detail - code-named as "GrepTheWeb".

Cloud Architecture Example: GrepTheWeb

The Alexa Web Search web service allows developers to build customized search engines against the massive data that Alexa crawls every night. One of the features of their web service allows users to query the Alexa search index and get Million Search Results (MSR) back as output. Developers can run queries that return up to 10 million results.

The resulting set, which represents a small subset of all the documents on the web, can then be processed further using a regular expression language. This allows developers to filter their search results using criteria that are *not* indexed by Alexa (Alexa indexes documents based on fifty different document attributes) thereby giving the developer power to do more sophisticated searches. Developers can run regular expressions against the actual documents, even when there are millions of them, to search for patterns and retrieve the subset of documents that matched that regular expression.

This application is currently in production at Amazon.com and is code-named *GrepTheWeb* because it can "grep" (a popular Unix command-line utility to search patterns) the actual web documents. GrepTheWeb allows developers to do some pretty specialized searches like selecting documents that have a particular HTML tag or META tag or finding documents with particular punctuations ("Hey!", he said. "Why Wait?"), or searching for mathematical equations ("f(x) = ∑x + W"), source code, e-mail addresses or other patterns such as "(dis)integration of life".

While the functionality is impressive, for us the way it was built is even more so. In the next section, we will zoom in to see different levels of the architecture of GrepTheWeb.

Figure 1 shows a high-level depiction of the architecture. The output of the Million Search Results Service, which is a sorted list of links and gzipped (compressed using the Unix gzip utility) in a single file, is given to GrepTheWeb as input. It takes a regular expression as a second input. It then returns a filtered subset of document links sorted and gzipped into a single file. Since the overall process is asynchronous, developers can get the status of their jobs by calling GetStatus() to see whether the execution is completed.

Performing a regular expression against millions of documents is not trivial. Different factors could combine to cause the processing to take lot of time:

- Regular expressions could be complex
- Dataset could be large, even hundreds of terabytes
- Unknown request patterns, e.g., any number of people can access the application at any given point in time

Hence, the design goals of GrepTheWeb included to scale in all dimensions (more powerful pattern-matching languages, more concurrent users of common datasets, larger datasets, better result qualities) while keeping the costs of processing down.

The approach was to build an application that not only scales with demand, but also without a heavy upfront investment and without the cost of maintaining idle machines ("downbottom"). To get a response in a reasonable amount of time, it was important to distribute the job into multiple tasks and to perform a Distributed Grep operation that runs those tasks on multiple nodes in parallel.

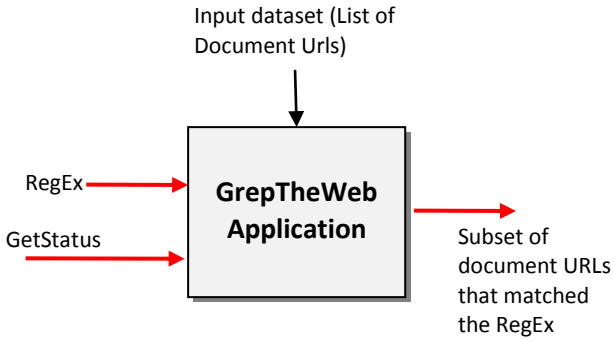


Figure 1 : GrepTheWeb Architecture - Zoom Level 1

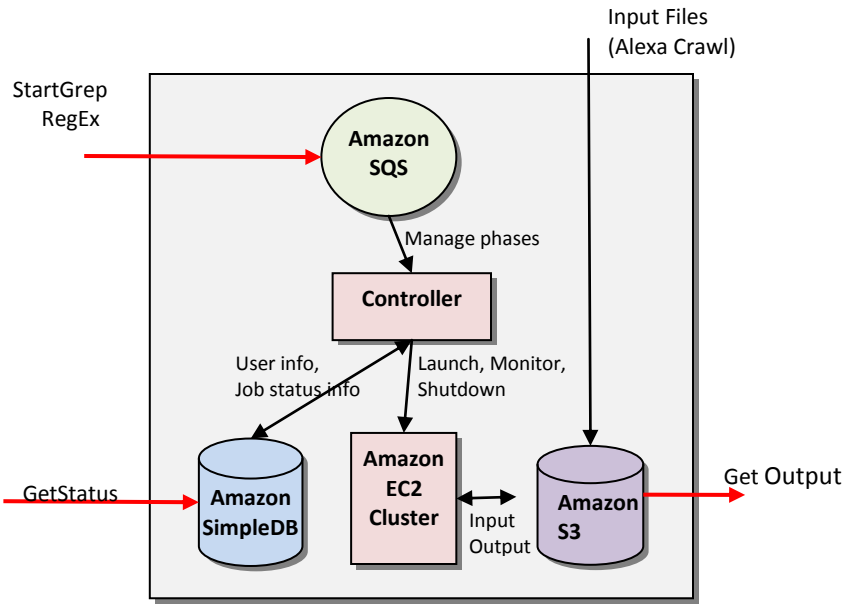


Figure 2: GrepTheWeb Architecture - Zoom Level 2

Zooming in further, GrepTheWeb architecture looks like as shown in Figure 2 (above). It uses the following AWS components:

- **Amazon S3** for retrieving input datasets and for storing the output dataset
- **Amazon SQS** for durably buffering requests acting as a "glue" between controllers
- **Amazon SimpleDB** for storing intermediate status, log, and for user data about tasks
- **Amazon EC2** for running a large distributed processing Hadoop cluster on-demand
- **Hadoop** for distributed processing, automatic parallelization, and job scheduling

Workflow

GrepTheWeb is modular. It does its processing in four phases as shown in figure 3. The launch phase is responsible for validating and initiating the processing of a GrepTheWeb request, instantiating Amazon EC2 instances, launching the Hadoop cluster on them and starting all the job processes. The monitor phase is responsible for monitoring the EC2 cluster, maps, reduces, and checking for success and failure. The shutdown phase is responsible for billing and shutting down all Hadoop processes and Amazon EC2 instances, while the cleanup phase deletes Amazon SimpleDB transient data.

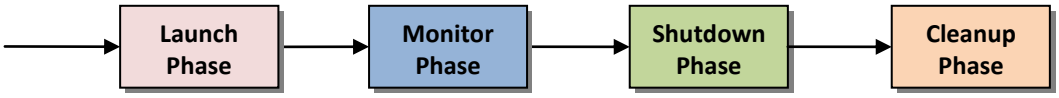


Figure 3: Phases of GrepTheWeb Architecture

Detailed Workflow for Figure 4:

1. On application start, queues are created if not already created and all the controller threads are started. Each controller thread starts polling their respective queues for any messages.
2. When a StartGrep user request is received, a launch message is enqueued in the launch queue.
3. *Launch phase*: The launch controller thread picks up the launch message, and executes the launch task, updates the status and timestamps in the Amazon SimpleDB domain, enqueues a new message in the monitor queue and deletes the message from the launch queue after processing.
    a. The launch task starts Amazon EC2 instances using a JRE pre-installed AMI , deploys required Hadoop libraries

and starts a Hadoop Job (run Map/Reduce tasks).

b. Hadoop runs map tasks on Amazon EC2 slave nodes in parallel. Each map task takes files (multithreaded in background) from Amazon S3, runs a regular expression (Queue Message Attribute) against the file from Amazon S3 and writes the match results along with a description of up to 5 matches locally and then the combine/reduce task combines and sorts the results and consolidates the output.

c. The final results are stored on Amazon S3 in the output bucket

4. *Monitor phase:* The monitor controller thread picks up this message, validates the status/error in Amazon SimpleDB and executes the monitor task, updates the status in the Amazon SimpleDB domain, enqueues a new message in the shutdown queue and billing queue and deletes the message from monitor queue after processing.
   a. The monitor task checks for the Hadoop status (JobTracker success/failure) in regular intervals, updates the SimpleDB items with status/error and Amazon S3 output file.
5. *Shutdown phase*: The shutdown controller thread picks up this message from the shutdown queue, and executes the shutdown task, updates the status and timestamps in Amazon SimpleDB domain, deletes the message from the shutdown queue after processing.
   a. The shutdown task kills the Hadoop processes, terminates the EC2 instances after getting EC2 topology information from Amazon SimpleDB and disposes of the infrastructure.
   b. The billing task gets EC2 topology information, Simple DB Box Usage, Amazon S3 file and query input and calculates the billing and passes it to the billing service.
6. *Cleanup phase*: Archives the SimpleDB data with user info.
7. Users can execute GetStatus on the service endpoint to get the status of the overall system (all controllers and Hadoop) and download the filtered results from Amazon S3 after completion.

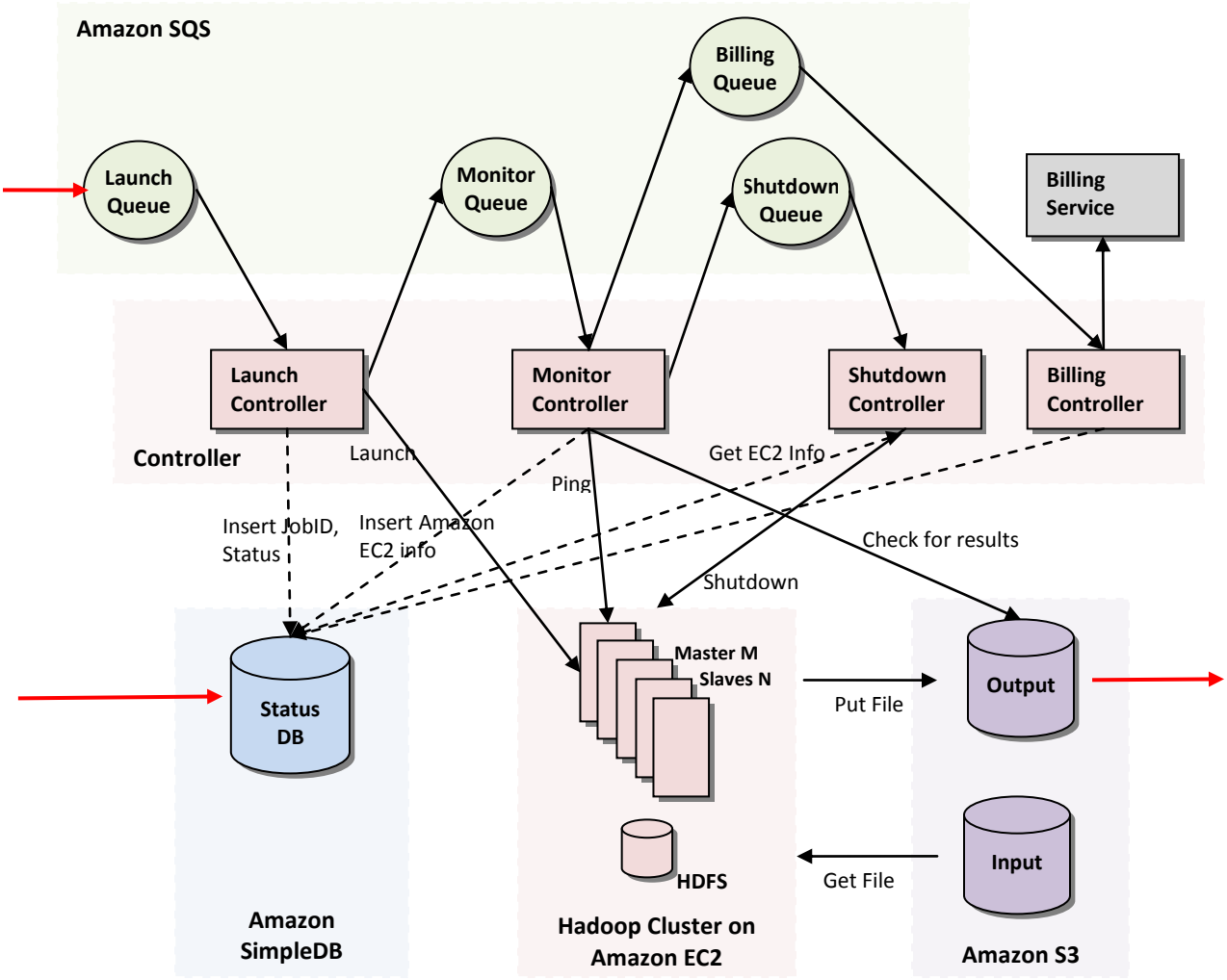


Figure 4: GrepTheWeb Architecture - Zoom Level 3

The Use of Amazon Web Services

In the next four subsections we present rationales of use and describe how GrepTheWeb uses AWS services.

How Was Amazon S3 Used

In GrepTheWeb, Amazon S3 acts as an input as well as an output data store. The input to GrepTheWeb is the web itself (compressed form of Alexa's Web Crawl), stored on Amazon S3 as objects and updated frequently. Because the web crawl dataset can be huge (usually in terabytes) and always growing, there was a need for a distributed, bottomless persistent storage. Amazon S3 proved to be a perfect fit.

How Was Amazon SQS Used

Amazon SQS was used as message-passing mechanism between components. It acts as "glue" that wired different functional components together. This not only helped in making the different components loosely coupled, but also helped in building an overall more failure resilient system.

Buffer

If one component is receiving and processing requests faster than other components (an unbalanced producer consumer situation), buffering will help make the overall system more resilient to bursts of traffic (or load). Amazon SQS acts as a transient buffer between two components (controllers) of the GrepTheWeb system. If a message is sent directly to a component, the receiver will need to consume it at a rate dictated by the sender. For example, if the billing system was slow or if the launch time of the Hadoop cluster was more than expected, the overall system would slow down, as it would just have to wait. With message queues, sender and receiver are decoupled and the queue service smoothens out any "spiky" message traffic.

Isolation

Interaction between any two controllers in GrepTheWeb is through messages in the queue and no controller directly calls any other controller. All communication and interaction happens by storing messages in the queue (en-queue) and retrieving messages from the queue (de-queue). This makes the entire system loosely coupled and the interfaces simple and clean. Amazon SQS provided a uniform way of transferring information between the different application components. Each controller's function is to retrieve the message, process the message (execute the function) and store the message in other queue while they are completely isolated from others.

Asynchrony

As it was difficult to know how much time each phase would take to execute (e.g., the launch phase decides dynamically how many instances need to start based on the request and hence execution time is unknown) Amazon SQS helped in building asynchronous systems. Now, if the launch phase takes more time to process or the monitor phase fails, the other components of the system are not affected and the overall system is more stable and highly available.

How Was Amazon SimpleDB Used

One use for a database in Cloud Architectures is to track statuses. Since the components of the system are asynchronous, there is a need to obtain the status of the system at any given point in time. Moreover, since all components are autonomous and discrete there is a need for a query-able datastore that captures the state of the system.

Because Amazon SimpleDB is schema-less, there is no need to define the structure of a record beforehand. Every controller can define its own structure and append data to a "job" item. For example: For a given job, "run email address regex over 10 million documents", the launch controller will add/update the "launch_status" attribute along with the "launch_starttime", while the monitor controller will add/update the "monitor_status" and "hadoop_status" attributes with enumeration values (running, completed, error, none). A GetStatus() call will query Amazon SimpleDB and return the state of each controller and also the overall status of the system.

Component services can query Amazon SimpleDB anytime because controllers independently store their states–one more nice way to create asynchronous highly-available services. Although, a simplistic approach was used in implementing the use of Amazon SimpleDB in GrepTheWeb, a more sophisticated approach, where there was complete, almost real-time monitoring would also be possible. For example, storing the Hadoop JobTracker status to show how many maps have been performed at a given moment.

Amazon SimpleDB is also used to store active Request IDs for historical and auditing/billing purposes.

In summary, Amazon SimpleDB is used as a status database to store the different states of the components and a historical/log database for querying high performance data.

How Was Amazon EC2 Used

In GrepTheWeb, all the controller code runs on Amazon EC2 Instances. The launch controller spawns master and slave instances using a pre-configured Amazon Machine Image (AMI). Since the dynamic provisioning and decommissioning happens using simple web service calls, GrepTheWeb knows how many master and slave instances needs to be launched.

The launch controller makes an educated guess, based on reservation logic, of how many slaves are needed to perform a particular job. The reservation logic is based on the complexity of the query (number of predicates etc) and the size of the input dataset (number of documents to be searched). This was also kept configurable so that we can reduce the processing time by simply specifying the number of instances to launch.

After launching the instances and starting the Hadoop cluster on those instances, Hadoop will appoint a master and slaves, handles the negotiating, handshaking and file distribution (SSH keys, certificates) and runs the grep job.

Hadoop Map Reduce

Hadoop is an open source distributed processing framework that allows computation of large datasets by splitting the dataset into manageable chunks, spreading it across a fleet of machines and managing the overall process by launching jobs, processing the job no matter where the data is physically located and, at the end, aggregating the job output into a final result.

It typically works in three phases. A *map* phase transforms the input into an intermediate representation of key value pairs, a *combine* phase (handled by Hadoop itself) combines and sorts by the keys and a *reduce* phase recombines the intermediate representation into the final output. Developers implement two interfaces, Mapper and Reducer, while Hadoop takes care of all the distributed processing (automatic parallelization, job scheduling, job monitoring, and result aggregation).

In Hadoop, there's a master process running on one node to oversee a pool of slave processes (also called workers) running on separate nodes. Hadoop splits the input into chunks. These chunks are assigned to slaves, each slave performs the map task (logic specified by user) on each pair found in the chunk and writes the results locally and informs the master of the completed status. Hadoop combines all the results and sorts the results by the keys. The master then assigns keys to the reducers. The reducer pulls the results using an iterator, runs the reduce task (logic specified by user), and sends the "final" output back to distributed file system.

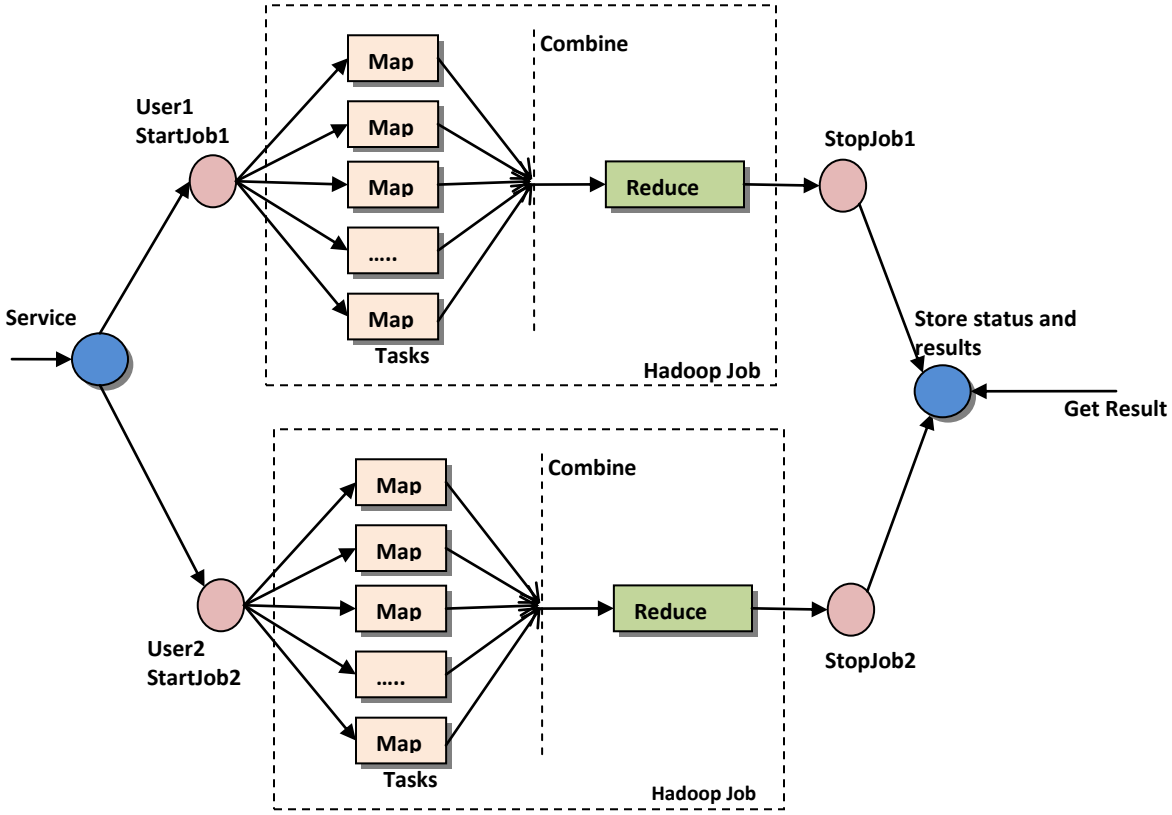


Figure 5: Map Reduce Operation (in GrepTheWeb)

GrepTheWeb Hadoop Implementation

Hadoop suits well the GrepTheWeb application. As each grep task can be run in parallel independently of other grep tasks using the parallel approach embodied in Hadoop is a perfect fit.

For GrepTheWeb, the actual documents (the web) are crawled ahead of time and stored on Amazon S3. Each user starts a grep job by calling the StartGrep function at the service endpoint. When triggered, masters and slave nodes (Hadoop cluster) are started on Amazon EC2 instances. Hadoop splits the input (document with pointers to Amazon S3 objects) into multiple manageable chunks of 100 lines each and assign the chunk to a slave node to run the map task. The map task reads these lines and is responsible for fetching the files from Amazon S3, running the regular expression on them and writing the results locally. If there is no match, there is no output. The map tasks then passes the results to the reduce phase which is an identity function (pass through) to aggregate all the outputs. The "final" output is written back to Amazon S3.

Example

Regular Expression
"A(.*)zon"

Format of the line in the Input dataset
[URL] [Title] [charset] [size] [S3 Object Key of .gz file] [offset]

http://www.amazon.com/gp/browse.html?node=3435361 Amazon Web us-ascii 3509 /2008/01/08/51/1/51_1_20080108072442_crawl100.arc.gz 70150864

Mapper Implementation

1. Key = line number and value = line in the input dataset
2. Create a signed URL (using Amazon AWS credentials) using the contents of key-value
3. Read (fetch) Amazon S3 Object (file) into a buffer
4. Run regular expression on that buffer
5. If there is match, collect the output in new set of key-value pairs (key = line, value = up to 5 matches)

Reducer Implementation - Pass-through (Built-in Identity Function) and write the results back to S3.

Tips for Designing a Cloud Architecture Application

1. Ensure that your application is scalable by designing each **component** to be scalable on its own. If every component implements a service interface, **responsible for its own scalability** in all appropriate dimensions, then the overall system will have a scalable base.
2. For better manageability and high-availability, make sure that your components are **loosely coupled**. The key is to build components without having tight dependencies between each other, so that if one component were to die (fail), sleep (not respond) or remain busy (slow to respond) for some reason, the other components in the system are built so as to continue to work as if no failure is happening.
3. Implement **parallelization** for better use of the infrastructure and for performance. Distributing the tasks on multiple machines, multithreading your requests and effective aggregation of results obtained in parallel are some of the techniques that help exploit the infrastructure.
4. After designing the basic functionality, ask the question "What if this fails?" Use techniques and approaches that will ensure **resilience**. If any component fails (and failures happen all the time), the system should automatically alert, failover, and re-sync back to the "last known state" as if nothing had failed.
5. Don't forget the **cost factor**. The key to building a cost-effective application is using on-demand resources in your design. It's wasteful to pay for infrastructure that is sitting idle.

Each of these points is discussed further in the context of GrepTheWeb.

Use Scalable Ingredients

The GrepTheWeb application uses highly-scalable components of the Amazon Web Services infrastructure that not only scale on-demand, but also are charged for on-demand.

All components of GrepTheWeb expose a service interface that defines the functions and can be called using HTTP requests and get back XML responses. For programming convenience small client libraries wrap and abstract the service specific code.

Each component is independent from the others and scales in all dimensions. For example, if thousands of requests hit Amazon SimpleDB, it can handle the demand because it is designed to handle massive parallel requests.

Likewise, distributed processing frameworks like Hadoop are designed to scale. Hadoop automatically distributes jobs, resumes failed jobs, and runs on multiple nodes to process terabytes of data.

Have Loosely Coupled Systems

The GrepTheWeb team built a loosely coupled system using *messaging queues*. If a queue/buffer is used to "wire" any two components together, it can support concurrency, high availability and load spikes. As a result, the overall system continues to perform even if parts of components become unavailable. If one component dies or becomes temporarily unavailable, the system will buffer the messages and get them processed when the component comes back up.

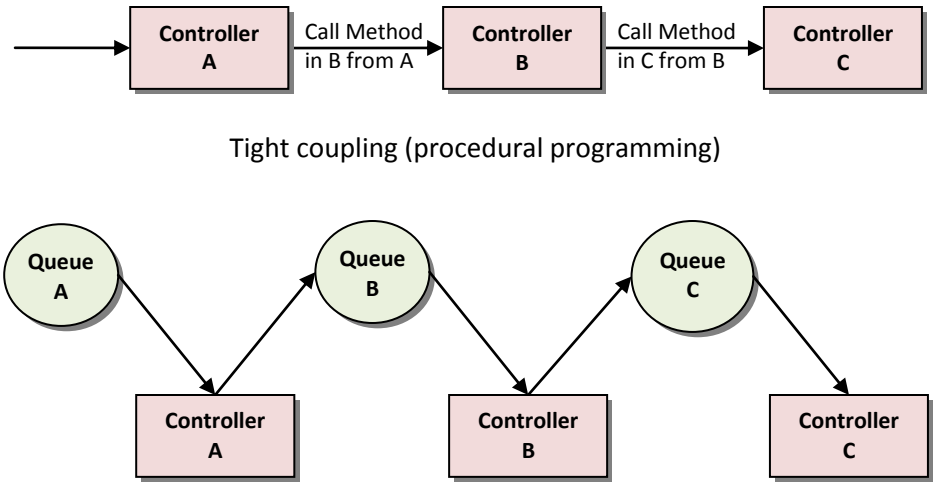


Figure 6: Loose Coupling – Independent Phases

In GrepTheWeb, for example, if lots of requests suddenly reach the server (an Internet-induced overload situation) or the processing of regular expressions takes a longer time than the median (slow response rate of a component), the Amazon SQS queues buffer the requests durably so those delays do not affect other components.

As in a multi-tenant system is important to get statuses of message/request, GrepTheWeb supports it. It does it by storing and updating the status of your each request in a separate query-able data store. This is achieved using Amazon SimpleDB. This combination of Amazon SQS for queuing and Amazon SimpleDB for state management helps achieve higher resilience by loose coupling.

Think Parallel

In this "era of tera" and multi-core processors, when programming we ought to think multi-*threaded* processes.

In GrepTheWeb, wherever possible, the processes were made thread-safe through a share-nothing philosophy and were multi-threaded to improve performance. For example, objects are fetched from Amazon S3 by multiple concurrent threads as such access is faster than fetching objects sequentially one at the time.

If multi-threading is not sufficient, think multi-*node*. Until now, parallel computing across large cluster of machines was not only expensive but also difficult to achieve. First, it was difficult to get the funding to acquire a large cluster of machines and then once acquired, it was difficult to manage and maintain them. Secondly, after it was acquired and managed, there were technical problems. It was difficult to run massively distributed tasks on the machines, store and access large datasets. Parallelization was not easy and job scheduling was error-prone. Moreover, if nodes failed, detecting them was difficult and recovery was very expensive. Tracking jobs and status was often ignored because it quickly became complicated as number of machines in cluster increased.

But now, computing has changed. With the advent of Amazon EC2, provisioning a large number of compute instances is easy. A cluster of compute instances can be provisioned within minutes with just a few API calls and decommissioned as easily. With the arrival of distributed processing frameworks like Hadoop, there is no need for high-caliber, parallel computing consultants to deploy a parallel application. Developers with no prior experience in parallel computing can implement a few interfaces in few lines of code, and parallelize the job without worrying about job scheduling, monitoring or aggregation.

On-Demand Requisition and Relinquishment

In GrepTheWeb each building-block component is accessible via the Internet using web services, reliably hosted in Amazon's datacenters and available on-demand. This means that the application can request more resources (servers, storage, databases, queues) or relinquish them whenever needed.

A beauty of GrepTheWeb is its almost-zero-infrastructure before and after the execution. The entire infrastructure is instantiated in the cloud triggered by a job request (grep) and then is returned back to the cloud, when the job is done. Moreover, during execution, it scales on-demand; i.e. the application scales elastically based on number of messages and the size of the input dataset, complexity of regular expression and so-forth.

For GrepTheWeb, there is reservation logic that decides how many Hadoop slave instances to launch based on the complexity of the regex and the input dataset. For

example, if the regular expression does not have many predicates, or if the input dataset has just 500 documents, it will only spawn 2 instances. However, if the input dataset is 10 million documents, it will spawn up to 100 instances.

Use Designs that Are Resilient to Reboot and Re-Launch

Rule of thumb: Be a pessimist when using Cloud Architectures; assume things will fail. In other words, always design, implement and deploy for automated recovery from failure.

In particular, assume that your hardware *will* fail. Assume that outages *will* occur. Assume that some disaster *will* strike your application. Assume that you *will* be slammed with more requests per second some day. By being pessimist, you end up thinking about recovery strategies during design time, which helps in designing an overall system better. For example, the following strategies can help in event of adversity:

1. Have a coherent backup and restore strategy for your data
2. Build process threads that resume on reboot
3. Allow the state of the system to re-sync by reloading messages from queues
4. Keep pre-configured and pre-optimized virtual images to support (2) and (3) on launch/boot

Good cloud architectures should be impervious to reboots and re-launches. In GrepTheWeb, by using a combination of Amazon SQS and Amazon SimpleDB, the overall controller architecture is more resilient. For instance, if the instance on which controller thread was running dies, it can be brought up and resume the previous state as if nothing had happened. This was accomplished by creating a pre-configured Amazon Machine Image, which when launched dequeues all the messages from the Amazon SQS queue and their states from the Amazon SimpleDB domain item on reboot.

If a task tracker (slave) node dies due to hardware failure, Hadoop reschedules the task on another node automatically. This fault-tolerance enables Hadoop to run on large commodity server clusters overcoming hardware failures.

Results and Costs

We ran several tests. Email Address Regular Expression was ran against 10 million documents. While 48 concurrent instances took 21 minutes to process, 92 concurrent instances took less than 6 min to process. This time includes instance launch time and start time of the Hadoop cluster. The total cost for 48 instances was around $5 and 92 instances was less than $10.

Conclusion

Instead of building your applications on fixed and rigid infrastructures, Cloud Architectures provide a new way to build applications on on-demand infrastructures.

GrepTheWeb demonstrates how such applications can be built.

Without having any upfront investment, we were able to run a job massively distributed on multiple nodes in parallel and scale incrementally based on the demand (users, size of the input dataset). With no idle time, the application infrastructure was never underutilized.

In the next section, we will learn how each of the Amazon Infrastructure Service (Amazon EC2, Amazon S3, Amazon SimpleDB and Amazon SQS) was used and we will share with you some of the lessons learned and some of the best practices.

Best Practices from Lessons Learned

In this section we highlight some of the best practices from the lessons learned during implementation of GrepTheWeb.

Best Practices of Amazon S3

Upload Large Files, Retrieve Small Offsets

End-to-end transfer data rates in Amazon S3 are best when large files are stored instead of small tiny files (sizes in the lower KBs). So instead of storing individual files on Amazon S3, multiple files were bundled and compressed (gzip) into a blob (.gz) and then stored on Amazon S3 as objects. The individual files were retrieved using the standard HTTP GET request by providing a URL (bucket and key), offset (byte-range), and size (byte-length). As a result, the overall cost of storage was reduced due to reduction in the overall size of the dataset (because of compression) and consequently the lesser number of PUT requests required than otherwise.

Sort the Keys and Then Upload Your Dataset

Amazon S3 reconcilers show performance improvement if the keys are pre-sorted before upload. By running a small script, the keys (URL pointers) were sorted and then uploaded in sorted order to Amazon S3.

Use Multi-threaded Fetching

Instead of fetching objects one by one from Amazon S3, multiple concurrent fetch threads were started within each map task to fetch the objects. However, it is not advisable to spawn 100s of threads because every node has bandwidth constraints. Ideally, users should try slowly ramping up their number of concurrent parallel threads until they find the point where adding additional threads offers no further speed improvement.

Use Exponential Back-off and Then Retry

A reasonable approach for any application is to retry every failed web service request. What is not obvious is what strategy to use to determine the retry interval. Our recommended approach is to use the truncated binary exponential back-off. In this approach the exact time to sleep between retries is determined by a combination of successively doubling the number of seconds that the maximum delay may be and choosing randomly a value in that range.

We recommended that you build the exponential back-off, sleep, and retry logic into the error handling code of your client. Exponential back-off reduces the number of requests made to Amazon S3 and thereby reduces the overall cost, while not overloading any part of the system.

Best Practices of Amazon SQS

Store Reference Information in the Message

Amazon SQS is ideal for small short-lived messages in workflows and processing pipelines. To stay within the message size limits it is advisable to store reference information as a part of the message and to store the actual input file on Amazon S3.

In GrepTheWeb, the launch queue message contains the URL of the input file (.dat.gz) which is a small subset of a result set (Million Search results that can have up to 10 million links). Likewise, the shutdown queue message contains the URL of the output file (.dat.gz), which is a filtered result set containing the links which match the regular expression.

The following tables show the message format of the queue and their statuses

ActionRequestId	f474b439-ee32-4af0-8e0f-a62d1f7de897
Code	Queued
Message	Your request has been queued.
ActionName	StartGrep
RegEx	A(.*)zon
InputUrl	http://s3.amazonaws.com/com.alexa.msr.prod/msr_f474b439-ee32-4af0-8e0f-a979907de897.dat.gz?Signature=CvD9iHA%3D&Expires=1204840434&AWSAccessKeyId=DDXCXCCDEEDSDFGSDDX

ActionRequestId	f474b439-ee32-4af0-8e0f-a62d1f7de897
Code	Completed
Message	Results are now available for download from DownloadUrl
ActionName	StartGrep
StartDate	2008-03-05T12:33:05
DownloadUrl	http://s3.amazonaws.com/com.alexa.gtw.prod/gtw_f474b439-ee32-4af0-8e0f-a62de897.dat.gz?Signature=CvD9iIGGjUIlkOlAeHA%3D&Expires=1204840434&AWSAccessKeyId=DDXCXCCDEEDSDFGSDDX

Use Process-oriented Messaging and Document-oriented Messaging

There are two messaging approaches that have worked effectively for us: process oriented and document oriented messaging. Process-oriented messaging is often defined by process or actions. The typical approach is to delete the old message from the "from" queue, and then to add a new message with new attributes to the new "to" queue.

Document-oriented messaging happens when one message per user/job thread is passed through the entire system with different message attributes. This is often implemented using XML/JSON because it has an extensible model. In this solution, messages can evolve, except that the receiver only needs to understand those parts that are important to him. This way a single message can flow through the system and the different

components only need to understand the parts of the message that is important to them.

For GrepTheWeb, we decided to use the process-oriented approach.

Take Advantage Of Visibility Timeout Feature

Amazon SQS has a special functionality that is not present in many other messaging systems; when a message is read from the queue it is visible to other readers of the queue yet it is not automatically deleted from the queue. The consumer needs to explicitly delete the message from the queue. If this hasn't happened within a certain period after the message was read, the consumer is considered to have failed and the message will re-appear in the queue to be consumed again. This is done by setting the so-called visibility timeout when creating the queue. In GrepTheWeb, the visibility timeout is very important because certain processes (such as the shutdown controller) might fail and not respond (e.g., instances would stay up). With the visibility timeout set to a certain number of minutes, another controller thread would pick up the old message and resume the task (of shutting down).

Best practices of Amazon SimpleDB

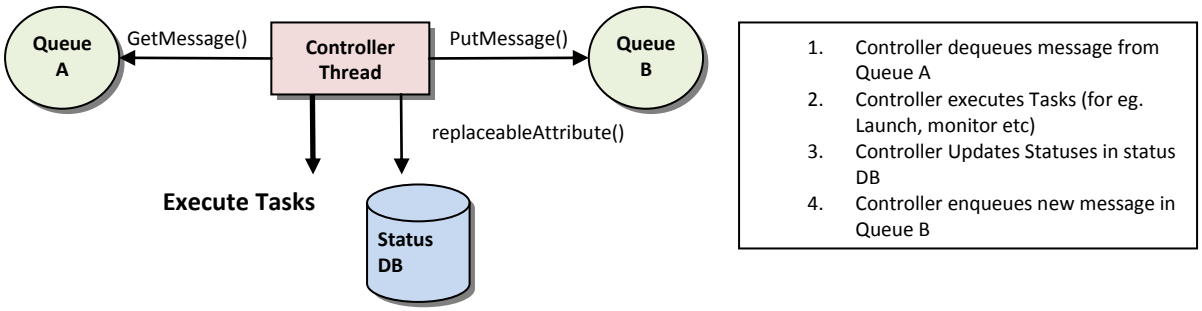
Multithread GetAttributes() and PutAttributes()

In Amazon SimpleDB, domains have items, and items have attributes. Querying Amazon SimpleDB returns a set of items. But often, attribute values are needed to perform a particular task. In that case, a query call is followed by a series of GetAttributes calls to get the attributes of each item in the list. As you can guess, the execution time would be slow. To address this, it is highly recommended to multi-thread your GetAttributes calls and to run them in parallel. The overall performance increases dramatically (up to 50 times) when run in parallel. In the GrepTheWeb application to generate monthly activity reports, this approach helped create more dynamic reports.

Use Amazon SimpleDB in Conjunction With Other Services

Build frameworks, libraries and utilities that use functionality of two or more services together in one. For GrepTheWeb, we built a small framework that uses Amazon SQS and Amazon SimpleDB together to externalize appropriate state. For example, all controllers are inherited from the BaseController class. The BaseController class's main responsibility is to dequeue the message from the "from" queue, validate the statuses from a particular Amazon SimpleDB domain, execute the function, update the statuses with a new timestamp and status, and put a new message in the "to" queue. The advantage of such a setup is that in an event of hardware failure or when controller instance dies, a new node can be brought up almost immediately and resume the state of operation by getting the messages back from the Amazon SQS queue and their status from Amazon SimpleDB upon reboot and makes the overall system more resilient.

Although not used in this design, a common practice is to store actual files as objects on Amazon S3 and to store all the metadata related to the object on Amazon SimpleDB. Also, using an Amazon S3 key to the object as item name in Amazon SimpleDB is a common practice.




```
Public Abstract BaseController (SQSMessageQueue fromQueue, SQSMessageQueue toQueue, SDBDomain
domain)
```

Figure 7: Controller Architecture and Workflow

Best Practices of Amazon EC2

Launch Multiple Instances All At Once

Instead of waiting for your EC2 instances to boot up one by one, we recommend that you start all of them at once with a simple run-instances command that specifies the number of instances of each type.

Automate As Much As Possible

This is applicable in everything we do and requires a special mention because automation of Amazon EC2 is often ignored. One of the biggest features of Amazon EC2 is that you can provision any number of compute instances by making a simple web service call. Automation will empower the developer to run a dynamic programmable datacenter that expands and contracts based on his needs. For example, automating your build-test-deploy cycle in the form of an Amazon Machine Image (AMI) and then running it automatically on Amazon EC2 every night (using a CRON job) will save a lot of time. By automating the AMI creation process, one can save a lot of time in configuration and optimization.

Add Compute Instances On-The-Fly

With Amazon EC2, we can fire up a node within minutes. Hadoop supports the dynamic addition of new nodes and task tracker nodes to a running cluster. One can simply launch new compute instances and start Hadoop processes on them, point them to the master and dynamically grow (and shrink) the cluster in real-time to speed up the overall process.

Safeguard Your AWS credentials When Bundling an AMI

If your AMI is running processes that need to communicate with other AWS web services (for polling the Amazon SQS queue or for reading objects from Amazon S3), one common design mistake is embedding the AWS credentials in the AMI. Instead of embedding the credentials, they should be passed in as arguments using the parameterized launch feature and encrypted before being sent over the wire. General steps are:

1. Generate a new RSA keypair (use OpenSSL tools).
2. Copy the private key onto the image, before you bundle it (so it will be embedded in the final AMI).
3. Post the public key along with the image details, so users can use it.
4. When a user launches the image they must first encrypt their AWS secret key (or private key if you wanted to use SOAP) with the public key you gave them in step 3. This encrypted data should be injected via user-data at launch (i.e. the parameterized launch feature).
5. Your image can then decrypt this at boot time and use it to decrypt the data required to contact Amazon S3. Also be sure to delete this private key upon reboot before installing the SSH key (i.e. before users can log into the machine). If users won't have root access then you don't have to delete the private key, just make sure it's not readable by users other than root.

Credits

Special Thanks to Kenji Matsuoka and Tinou Bao – the core team that developed the GrepTheWeb Architecture.

Further Reading

Amazon SimpleDB White Papers
Amazon SQS White paper
Hadoop Wiki
Hadoop Website
Distributed Grep Examples
Map Reduce Paper

Blog: Taking Massive Distributed Computing to the Common man – Hadoop on Amazon EC2/S3

Appendix 1: Amazon S3, Amazon SQS, Amazon SimpleDB – When to Use Which?

The table will help explain which Amazon service to use when:

	Amazon S3	Amazon SQS	Amazon SimpleDB
Ideal for	Storing Large write-once, read-many types of objects	Small short-lived transient messages	Querying light-weight attribute data
Ideal examples	Media-like files, audio, video, large images	Workflow jobs, XML/JSON/TXT messages	Querying, Mapping, tagging, click-stream logs, metadata, state management
Not recommended for	Querying, content distribution	Large objects, persistent objects	Transactional systems
Not recommended examples	Database, File Systems	Persistent data stores	OLTP, DW cube rollups

Recommendations

Since the Amazon Web Services are primitive building block services, the most value is derived when they are used in conjunction with other services

- **Use Amazon S3 and Amazon SimpleDB together whenever you want to query Amazon S3 objects using their metadata**

  We recommend you store large files on Amazon S3 and the associated metadata and reference information on Amazon SimpleDB so that developers can query the metadata. Read-only metadata can also be stored on Amazon S3 as metadata on object (e.g. author, create date etc).

Amazon S3 entities	Amazon SimpleDB entities
Bucket	Domain (private to subscriber)
Key/S3 URI	Item name
Metadata describing S3 object	Attributes of an item

- **Use SimpleDB and Amazon SQS together whenever you want an application to be in phases**

  Store transient messages in Amazon SQS and statuses of job/messages in Amazon SimpleDB so that you can update statuses frequently and get the status of any request at any time by simply querying the item. This works especially well in asynchronous systems.

- **Use Amazon S3 and Amazon SQS together whenever you want to create processing pipelines or producer-consumer solutions**

  Store raw files on Amazon S3 and insert a corresponding message in an Amazon SQS queue with reference and metadata (S3 URI etc)