



Building Applications

— with —

DynamoDB

An Online Seminar - 16th May 2012

Dr Matt Wood, Amazon Web Services

Thank you!

Building Applications with DynamoDB



Building Applications with DynamoDB



Getting started



Building Applications with DynamoDB



Getting started



Data modeling



Building Applications with DynamoDB



Getting started



Data modeling



Partitioning



Building Applications with DynamoDB



Getting started



Data modeling



Partitioning



Analytics



Getting started with DynamoDB

QUICK REVIEW



DynamoDB is a managed
NoSQL database service.

Store and retrieve any amount of data.
Serve any level of request traffic.



Without the
operational burden.



Consistent, predictable
performance.

Single digit millisecond latencies.
Backed on solid-state drives.



Flexible data model.

Key/attribute pairs.

No schema required.

Easy to create. Easy to adjust.



Seamless scalability.

No table size limits. Unlimited storage.

No downtime.



Durable.

Consistent, disk-only writes.

Replication across data centres and
availability zones.



Without the
operational burden.



Without the
operational burden.

FOCUS ON YOUR APP



Two decisions + three clicks
= ready for use



PRIMARY KEYS +
LEVEL OF THROUGHPUT

Two decisions + three clicks
= ready for use



Provisioned throughput.

Reserve IOPS for reads and writes.

Scale up (or down) at any time.



Pay per capacity unit.

Priced per hour of
provisioned throughput.



Write throughput.

Units = size of item x writes/second

\$0.01 per hour for 10 write units



Consistent writes.

Atomic increment/decrement.

Optimistic concurrency control.

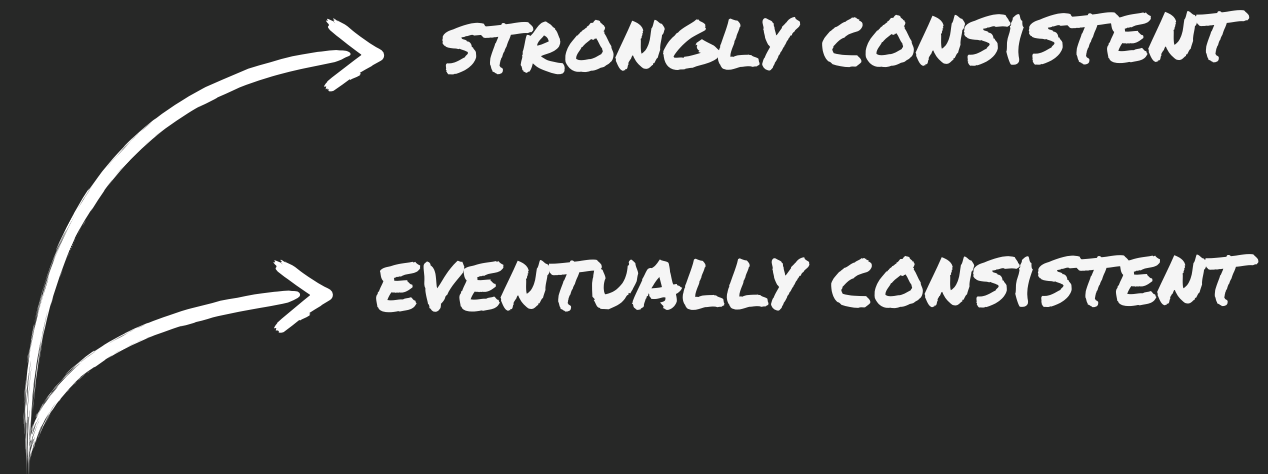
aka: "conditional writes".



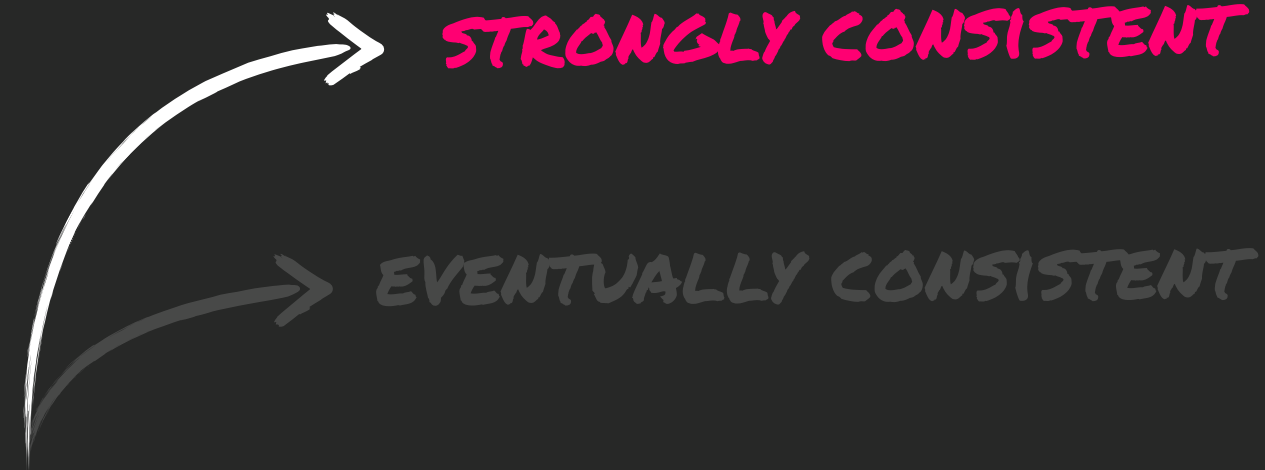
Transactions.

Item level transactions only.

Puts, updates and deletes are ACID.



Read throughput.



Read throughput.

Provisioned units =
size of item x reads/second

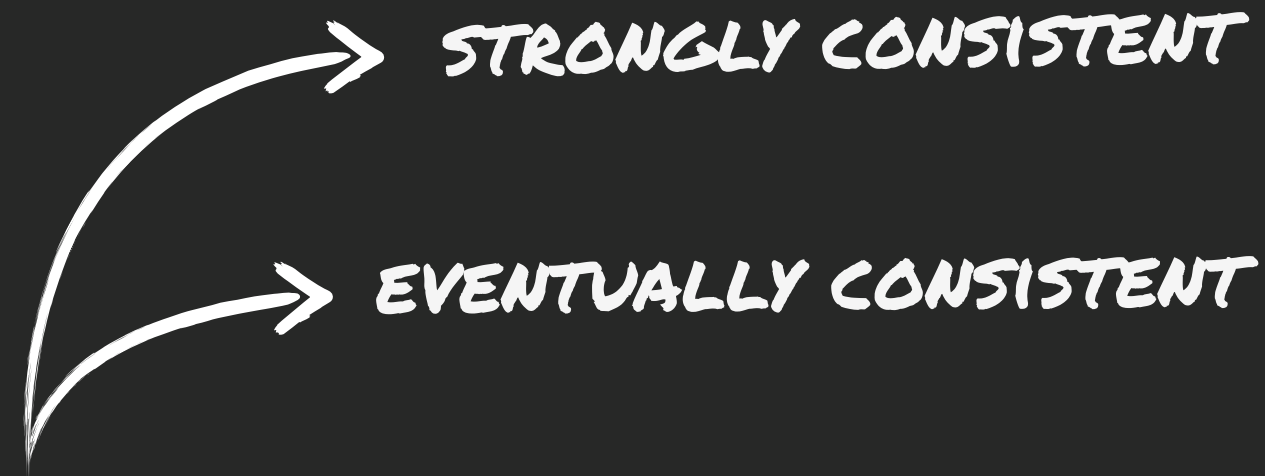
\$0.01 per hour for 50 read units



Read throughput.

$$\text{Provisioned units} = \frac{\text{size of item} \times \text{reads/second}}{2}$$

\$0.01 per hour for 100 read units



Read throughput.

Same latency expectations.

Mix and match at "read time".



Two decisions + three clicks
= ready for use

←→↻

https://console.aws.amazon.com/dynamodb/home

☆🔧

AWS Management Console > Amazon DynamoDB

Matt Wood | Help

AWS Elastic Beanstalk

Amazon S3

Amazon EC2

Amazon VPC

Amazon CloudWatch

Amazon Elastic MapReduce

Amazon CloudFront

AWS CloudFormation

Amazon RDS

Amazon ElastiCache

Amazon SQS

AWS IAM

Amazon SNS

Amazon SES

Amazon Route 53

More...

Tables

Region: 🇺🇸 US East (N. Virginia)

Filter:

▲ Name

images

items

tags

1 to 3 of 3 tables >>

Write Throughput

5

5

5

Create Table

Cancel

PRIMARY KEY

PROVISIONED THROUGHPUT CAPACITY

THROUGHPUT ALARMS (optional)

Table Name:

photos

Table will be created in us-east-1 region

Primary Key:

DynamoDB is a schema-less database. You only need to tell us your primary key attribute(s).

Primary Key Type: ☒ Hash ☐ Hash and Range

☒ String ☐ Number

Hash Attribute Name: id

⚠ Choose a hash attribute that ensures that your workload is evenly distributed across hash keys.

For example, "Customer ID" is a good hash key, while "Game ID" would be a bad choice if most of your traffic relates to a few popular games.

[Learn more about choosing your primary key](#)

Cancel

Continue

Help

© 2008 - 2012, Amazon Web Services LLC or its affiliates. All rights reserved.

Feedback

Support

Privacy Policy

Terms of Use

An amazon.com company

1

2

3

AWS Management Console

✕

←

→

↺

https://console.aws.amazon.com/dynamodb/home

☆

📄

🔧

AWS Management Console > Amazon DynamoDB

Matt Wood ▾ | Help ▾

📦

AWS

Elastic Beanstalk

Amazon

S3

Amazon

EC2

Amazon

VPC

Amazon

CloudWatch

Amazon

Elastic MapReduce

Amazon

CloudFront

AWS

CloudFormation

Amazon

RDS

Amazon

ElastiCache

Amazon

SQS

AWS

IAM

Amazon

SNS

Amazon

SES

Amazon

Route 53

More... ▾

Tables

Region: US East (N. Virginia)

Filter:

▲

Name

images

items

tags

1 to 3 of 3 tables > >

Write Throughput

5

5

5

Create Table

Cancel ✕

✓

PRIMARY KEY

PROVISIONED THROUGHPUT CAPACITY

THROUGHPUT ALARMS (optional)

Provisioned Throughput Capacity:

☐ Help me calculate how much throughput capacity I need to provision

Throughput capacity to provision:

Amazon DynamoDB lets you specify how much read and write throughput capacity you wish to provision for your table. Using this information, Amazon will provision the appropriate resources to meet your throughput needs. [More Information](#)

Read Capacity Units:

Write Capacity Units:

⚠️ If you exceed the free tier you are charged for the provisioned throughput capacity of your table **even if you do not actively use your provisioned capacity**. [Learn more about DynamoDB's free tier and pricing.](#)

Back

Continue ▶

Help 📄

© 2008 - 2012, Amazon Web Services LLC or its affiliates. All rights reserved.

Feedback

Support

Privacy Policy

Terms of Use

An amazon.com company

Region: US East (N. Virginia) ▼

images

items

tags

Cancel

PROVISIONED THROUGHPUT CAPACITY

THROUGHPUT ALARMS (optional)

☒ **Use Basic Alarms**

Notify me when my table's request rates exceed 80% of Provisioned Throughput for 60 minutes.

Notification will be sent when:

- Read Capacity Units consumed > 4
or
- Write Capacity Units consumed > 4

Send notification to:

matthew@amazon.com

Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Service.

Advanced alarm settings are available in the CloudWatch Management Console.

[Back](#)

Create Table 

Help 



Two decisions + three clicks
= ready for use



Two decisions + one API call
= ready for use



```
$create_response = $dynamodb->create_table(array(  
    'TableName' => 'ProductCatalog',  
    'KeySchema' => array(  
        'HashKeyElement' => array(  
            'AttributeName' => 'Id',  
            'AttributeType' => AmazonDynamoDB::TYPE_NUMBER  
        )  
    ),  
    'ProvisionedThroughput' => array(  
        'ReadCapacityUnits' => 10,  
        'WriteCapacityUnits' => 5  
    )  
));
```



Two decisions + one API call
= ready for **use**



Two decisions + one API call
= ready for development



Two decisions + one API call
= ready for **production**



Two decisions + one API call
= ready for **scale**

Tables

Region: US East (N. Virginia)

Filter: tags

▲ Name

tags

Modify Provisioned Throughput

Cancel

Modify throughput for: **tags**

Provisioned Throughput

Throughput can be increased by a factor of two or less each time you modify your table.

Read Capacity Units: (allowed values: 5 - 20)

Write Capacity Units: (allowed values: 5 - 20)

☐ Use Basic Alarms

Notify me when my table's request rates exceed of Provisioned Throughput for 60 minutes.

Notification will be sent when:

- Read Capacity Units consumed > 16 or
- Write Capacity Units consumed > 16

Send notification to:

Cancel

Save

Table Items

Details

Alarms and Usage

Recent Alarms

-- No alarms triggered -- [Go to CloudWatch](#)

CloudWatch Metrics (for the past hour)

Click to view in CloudWatch. Links open in new windows or tabs

Table Name: tags

Metric Name:

Operation:



Authentication.

Session based to minimize latency.

Uses Amazon Security Token Service.

Handled by AWS SDKs.

Integrates with IAM.



Monitoring.

CloudWatch metrics:
latency, consumed read and write
throughput, errors and throttling.



Libraries, mappers & mocks.

ColdFusion, Django, Erlang, Java, .Net,
Node.js, Perl, PHP, Python, Ruby

<http://j.mp/dynamodb-lib>



DynamoDB data models



DynamoDB semantics.

Tables, items and attributes.



Tables contain items.

Unlimited items per table.



Items are a collection of attributes.

Each attribute has a key and a value.

An item can have any number of attributes, up to 64k total.



Two scalar data types.

String: Unicode, UTF8 binary encoding.

Number: 38 digit precision.

Multi-value strings and numbers.



id = 100	date = 2012-05-16-09-00-10	total = 25.00
id = 101	date = 2012-05-15-15-00-11	total = 35.00
id = 101	date = 2012-05-16-12-00-10	total = 100.00
id = 102	date = 2012-03-20-18-23-10	total = 20.00
id = 102	date = 2012-03-20-18-23-10	total = 120.00



TABLE

id = 100	date = 2012-05-16-09-00-10	total = 25.00
id = 101	date = 2012-05-15-15-00-11	total = 35.00
id = 101	date = 2012-05-16-12-00-10	total = 100.00
id = 102	date = 2012-03-20-18-23-10	total = 20.00
id = 102	date = 2012-03-20-18-23-10	total = 120.00



ITEM
→

id = 100	date = 2012-05-16-09-00-10	total = 25.00
id = 101	date = 2012-05-15-15-00-11	total = 35.00
id = 101	date = 2012-05-16-12-00-10	total = 100.00
id = 102	date = 2012-03-20-18-23-10	total = 20.00
id = 102	date = 2012-03-20-18-23-10	total = 120.00



ATTRIBUTE



id = 100	date = 2012-05-16-09-00-10	total = 25.00
id = 101	date = 2012-05-15-15-00-11	total = 35.00
id = 101	date = 2012-05-16-12-00-10	total = 100.00
id = 102	date = 2012-03-20-18-23-10	total = 20.00
id = 102	date = 2012-03-20-18-23-10	total = 120.00



Where is the schema?

Tables do not require a formal schema.
Items are an arbitrary sized hash.
Just need to specify the primary key.



Items are indexed by
primary key.

Single hash keys and composite keys.



HASH KEY



id = 100	date = 2012-05-16-09-00-10	total = 25.00
id = 101	date = 2012-05-15-15-00-11	total = 35.00
id = 101	date = 2012-05-16-12-00-10	total = 100.00
id = 102	date = 2012-03-20-18-23-10	total = 20.00
id = 102	date = 2012-03-20-18-23-10	total = 120.00



Range key for queries.

Querying items by composite key.



HASH KEY + RANGE KEY

id = 100	date = 2012-05-16-09-00-10	total = 25.00
id = 101	date = 2012-05-15-15-00-11	total = 35.00
id = 101	date = 2012-05-16-12-00-10	total = 100.00
id = 102	date = 2012-03-20-18-23-10	total = 20.00
id = 102	date = 2012-03-20-18-23-10	total = 120.00



Programming DynamoDB.

Small but perfectly formed.

Whole programming interface
fits on one slide.



CreateTable

PutItem

UpdateTable

GetItem

DeleteTable

UpdateItem

DescribeTable

DeleteItem

ListTables

BatchGetItem

Query

BatchWriteItem

Scan



CreateTable

PutItem

UpdateTable

GetItem

DeleteTable

UpdateItem

DescribeTable

DeleteItem

ListTables

BatchGetItem

Query

BatchWriteItem

Scan



CreateTable

PutItem

UpdateTable

GetItem

DeleteTable

UpdateItem

DescribeTable

DeleteItem

ListTables

BatchGetItem

Query

BatchWriteItem

Scan



Conditional updates.

PutItem, UpdateItem, DeleteItem can take optional conditions for operation.

UpdateItem performs atomic increments.



CreateTable

PutItem

UpdateTable

GetItem

DeleteTable

UpdateItem

DescribeTable

DeleteItem

ListTables

BatchGetItem

Query

BatchWriteItem

Scan



One API call, multiple items.

BatchGet returns multiple items by primary key.

BatchWrite performs up to 25 put or delete operations.

Throughput is measured by IO, not API calls.



CreateTable

PutItem

UpdateTable

GetItem

DeleteTable

UpdateItem

DescribeTable

DeleteItem

ListTables

BatchGetItem

Query

BatchWriteItem

Scan



Query vs Scan

Query for composite key queries.

Scan for full table scans, exports.

Both support pagination and limits.

Maximum response is 1Mb in size.



Query patterns.

Retrieve all items by hash key.

Range key conditions:

`==`, `<`, `>`, `>=`, `<=`, begins with, between.

Counts. Top and bottom n values.

Paged responses.



Modeling patterns



1. Mapping relationships with range keys.

No cross-table joins in DynamoDB.

Use composite keys to model relationships.

Data model example: online gaming.

Storing scores and leader boards.

**PLAYERS WITH
HIGH SCORES.**

**LEADER BOARD FOR
EACH GAME.**

Data model example: online gaming.

Storing scores and leader boards.

Players: hash key

user_id = mza	location = Cambridge	joined = 2011-07-04
user_id = jeffbarr	location = Seattle	joined = 2012-01-20
user_id = werner	location = Worldwide	joined = 2011-05-15

**PLAYERS WITH
HIGH SCORES.**

**LEADER BOARD FOR
EACH GAME.**

Data model example: online gaming.

Storing scores and leader boards.

Players: hash key

user_id = mza	location = Cambridge	joined = 2011-07-04
user_id = jeffbarr	location = Seattle	joined = 2012-01-20
user_id = werner	location = Worldwide	joined = 2011-05-15

Scores: composite key

user_id = mza	game = angry-birds	score = 11,000
user_id = mza	game = tetris	score = 1,223,000
user_id = werner	location = bejewelled	score = 55,000

**PLAYERS WITH
HIGH SCORES.**

**LEADER BOARD FOR
EACH GAME.**

Data model example: online gaming.

Storing scores and leader boards.

Players: hash key

user_id = mza	location = Cambridge	joined = 2011-07-04
user_id = jeffbarr	location = Seattle	joined = 2012-01-20
user_id = werner	location = Worldwide	joined = 2011-05-15

**PLAYERS WITH
HIGH SCORES.**

**LEADER BOARD FOR
EACH GAME.**

Scores: composite key

user_id = mza	game = angry-birds	score = 11,000
user_id = mza	game = tetris	score = 1,223,000
user_id = werner	location = bejewelled	score = 55,000

Leader boards: composite key

game = angry-birds	score = 11,000	user_id = mza
game = tetris	score = 1,223,000	user_id = mza
game = tetris	score = 9,000,000	user_id = jeffbarr

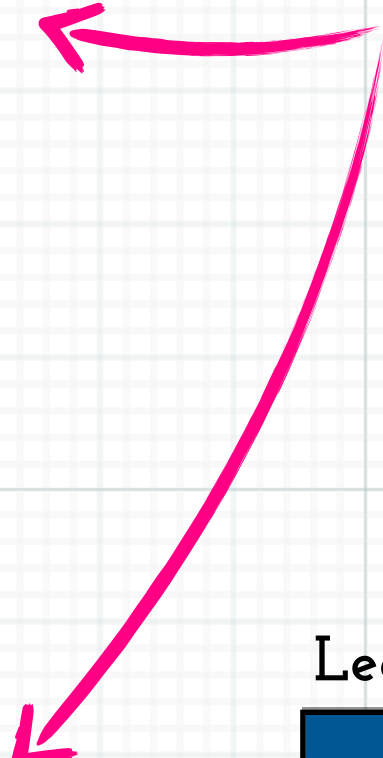
Data model example: online gaming.

Storing scores and leader boards.

Players: hash key

user_id = mza	location = Cambridge	joined = 2011-07-04
user_id = jeffbarr	location = Seattle	joined = 2012-01-20
user_id = werner	location = Worldwide	joined = 2011-05-15

Scores by user
(and by game)



Scores: composite key

user_id = mza	game = angry-birds	score = 11,000
user_id = mza	game = tetris	score = 1,223,000
user_id = werner	location = bejewelled	score = 55,000

Leader boards: composite key

game = angry-birds	score = 11,000	user_id = mza
game = tetris	score = 1,223,000	user_id = mza
game = tetris	score = 9,000,000	user_id = jeffbarr

Data model example: online gaming.

Storing scores and leader boards.

Players: hash key

user_id = mza	location = Cambridge	joined = 2011-07-04
user_id = jeffbarr	location = Seattle	joined = 2012-01-20
user_id = werner	location = Worldwide	joined = 2011-05-15

Scores: composite key

user_id = mza	game = angry-birds	score = 11,000
user_id = mza	game = tetris	score = 1,223,000
user_id = werner	location = bejewelled	score = 55,000

High scores by
game

Leader boards: composite key

game = angry-birds	score = 11,000	user_id = mza
game = tetris	score = 1,223,000	user_id = mza
game = tetris	score = 9,000,000	user_id = jeffbarr



2. Handling large items.

Unlimited attributes per item.

Unlimited items per table.

Max 64k per item.

Data model example: large items.

Storing more than 64k across items.

Large messages: composite keys

message_id = 1	part = 1	message = <first 64k>
message_id = 1	part = 2	message = <second 64k>
message_id = 1	part = 3	joined = <third 64k>

Split attributes across items.

Query by message_id and part to retrieve.



Store a pointer to objects in Amazon S3.

Large data stored in S3.

Location stored in DynamoDB.

99.99999999999999% data durability in S3.



3. Managing secondary indices.

Not supported by DynamoDB.

Create your own.

Data model example: secondary indices.

Storing more than 64k across items.

Users: hash key

user_id = mza	first_name = Matt	last_name = Wood
user_id = mattfox	first_name = Matt	last_name = Fox
user_id = werner	first_name = Werner	last_name = Vogels

Data model example: secondary indices.

Storing more than 64k across items.

Users: hash key

user_id = mza	first_name = Matt	last_name = Wood
user_id = mattfox	first_name = Matt	last_name = Fox
user_id = werner	first_name = Werner	last_name = Vogels

First name index: composite keys

first_name = Matt	user_id = mza
first_name = Matt	user_id = mattfox
first_name = Werner	user_id = werner

Data model example: secondary indices.

Storing more than 64k across items.

Users: hash key

user_id = mza	first_name = Matt	last_name = Wood
user_id = mattfox	first_name = Matt	last_name = Fox
user_id = werner	first_name = Werner	last_name = Vogels

First name index: composite keys

first_name = Matt	user_id = mza
first_name = Matt	user_id = mattfox
first_name = Werner	user_id = werner

Second name index: composite keys

last_name = Wood	user_id = mza
last_name = Fox	user_id = mattfox
last_name = Vogels	user_id = werner

Data model example: secondary indices.

Storing more than 64k across items.

Users: hash key

user_id = mza	first_name = Matt	last_name = Wood
user_id = mattfox	first_name = Matt	last_name = Fox
user_id = werner	first_name = Werner	last_name = Vogels

First name index: composite keys

first_name = Matt	user_id = mza
first_name = Matt	user_id = mattfox
first_name = Werner	user_id = werner

Second name index: composite keys

last_name = Wood	user_id = mza
last_name = Fox	user_id = mattfox
last_name = Vogels	user_id = werner

Data model example: secondary indices.

Storing more than 64k across items.

Users: hash key

user_id = mza	first_name = Matt	last_name = Wood
user_id = mattfox	first_name = Matt	last_name = Fox
user_id = werner	first_name = Werner	last_name = Vogels

First name index: composite keys

first_name = Matt	user_id = mza
first_name = Matt	user_id = mattfox
first_name = Werner	user_id = werner

Second name index: composite keys

last_name = Wood	user_id = mza
last_name = Fox	user_id = mattfox
last_name = Vogels	user_id = werner



4. Time series data.

Logging, click through, ad views,
game play data, application usage.

Non-uniform access patterns.

Newer data is 'live'.

Older data is read only.

Data model example: time series data.

Rolling tables for hot and cold data.

Events table: composite keys

event_id = 1000	timestamp = 2012-05-16-09-59-01	key = value
event_id = 1001	timestamp = 2012-05-16-09-59-02	key = value
event_id = 1002	timestamp = 2012-05-16-09-59-02	key = value

Data model example: time series data.

Rolling tables for hot and cold data.

Events table: composite keys

event_id = 1000	timestamp = 2012-05-16-09-59-01	key = value
event_id = 1001	timestamp = 2012-05-16-09-59-02	key = value
event_id = 1002	timestamp = 2012-05-16-09-59-02	key = value

Events table for April: composite keys

event_id = 400	timestamp = 2012-04-01-00-00-01
event_id = 401	timestamp = 2012-04-01-00-00-02
event_id = 402	timestamp = 2012-04-01-00-00-03

Events table for January: composite keys

event_id = 100	timestamp = 2012-01-01-00-00-01
event_id = 101	timestamp = 2012-01-01-00-00-02
event_id = 102	timestamp = 2012-01-01-00-00-03



Hot and cold tables.



Dec

Jan

Feb

Mar

April

May



Hot and cold tables.



Dec

Jan

Feb

Mar

April

May

HIGHER
THROUGHPUT

A white curved arrow pointing upwards and to the right, starting from the text 'HIGHER THROUGHPUT' and pointing towards the right end of the horizontal arrow.



Hot and cold tables.



Dec

Jan

Feb

Mar

April

May



LOWER

THROUGHPUT



HIGHER

THROUGHPUT



Hot and cold tables.



Dec

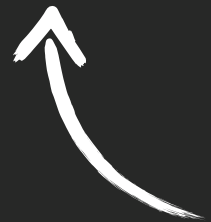
Jan

Feb

Mar

April

May



DATA TO S3,
DELETE COLD TABLES



Hot and cold tables.



Jan

Feb

Mar

Apr

May

June



Hot and cold tables.



Feb

Mar

Apr

May

June

July



Hot and cold tables.



Mar Apr May June July Aug



Hot and cold tables.



Apr

May

June

July

Aug

Sept



Hot and cold tables.



May June July Aug Sept Oct



Not out of mind.

DynamoDB and S3 data can be integrated for analytics.

Run queries across hot and cold data with Elastic MapReduce.



Partitioning best practices



Uniform workloads.

DynamoDB divides table data into multiple partitions.

Data is distributed primarily by hash key.

Provisioned throughput is divided evenly across the partitions.



Uniform workloads.

To achieve and maintain full provisioned throughput for a table, spread your workload evenly across the hash keys.



Non-uniform workloads.

Some requests might be throttled,
even at high levels of provisioned
throughput.

Some best practices...



1. Distinct values for hash keys.

Hash key elements should have a high number of distinct values.

Data model example: hash key selection.

Well distributed work loads

Users

user_id = mza	first_name = Matt	last_name = Wood
user_id = jeffbarr	first_name = Jeff	last_name = Barr
user_id = werner	first_name = Werner	last_name = Vogels
user_id = mattfox	first_name = Matt	last_name = Fox
...	...	...

Data model example: hash key selection.

Well distributed work loads

Users

user_id = mza	first_name = Matt	last_name = Wood
user_id = jeffbarr	first_name = Jeff	last_name = Barr
user_id = werner	first_name = Werner	last_name = Vogels
user_id = mattfox	first_name = Matt	last_name = Fox
...	...	...



Lots of users with unique user_id.

Workload well distributed across user partitions.



2. Avoid limited hash key values.

Hash key elements should have a high number of distinct values.

Data model example: small hash value range.

Non-uniform workload.

Status responses

status = 200	date = 2012-04-01-00-00-01
status = 404	date = 2012-04-01-00-00-01
status 404	date = 2012-04-01-00-00-01
status = 404	date = 2012-04-01-00-00-01

Data model example: small hash value range.

Non-uniform workload.

Status responses

status = 200	date = 2012-04-01-00-00-01
status = 404	date = 2012-04-01-00-00-01
status 404	date = 2012-04-01-00-00-01
status = 404	date = 2012-04-01-00-00-01



Small number of status codes.
Unevenly, non-uniform workload.



3. Model for even distribution of access.

Access by hash key value should be evenly distributed across the dataset.

Data model example: uneven access pattern by key.

Non-uniform access workload.

Devices

mobile_id = 100	access_date = 2012-04-01-00-00-01
mobile_id = 100	access_date = 2012-04-01-00-00-02
mobile_id = 100	access_date = 2012-04-01-00-00-03
mobile_id = 100	access_date = 2012-04-01-00-00-04
...	...

Data model example: uneven access pattern by key.

Non-uniform access workload.

Devices

mobile_id = 100	access_date = 2012-04-01-00-00-01
mobile_id = 100	access_date = 2012-04-01-00-00-02
mobile_id = 100	access_date = 2012-04-01-00-00-03
mobile_id = 100	access_date = 2012-04-01-00-00-04
...	...



Large number of devices.

Small number which are much more popular than others.

Workload unevenly distributed.

Data model example: randomize access pattern by key.

Towards a uniform workload.

Devices

mobile_id = 100.1	access_date = 2012-04-01-00-00-01
mobile_id = 100.2	access_date = 2012-04-01-00-00-02
mobile_id = 100.3	access_date = 2012-04-01-00-00-03
mobile_id = 100.4	access_date = 2012-04-01-00-00-04
...	...



Randomize access pattern.

Workload randomised by hash key.



Design for a uniform
workload.



Analytics with DynamoDB



Seamless scale.

Scalable methods for data processing.
Scalable methods for backup/restore.



Amazon Elastic MapReduce.

Managed Hadoop service for
data-intensive workflows.

<http://aws.amazon.com/emr>



Hadoop under the hood.

Take advantage of the Hadoop ecosystem: streaming interfaces, Hive, Pig, Mahout.



Distributed data processing.

API driven. Analytics at any scale.



Query flexibility with Hive.

```
create external table items_db
(id string, votes bigint, views bigint) stored by
'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
tblproperties
("dynamodb.table.name" = "items",
"dynamodb.column.mapping" =
"id:id,votes:votes,views:views");
```



Query flexibility with Hive.

```
select id, likes, views  
from items_db  
order by views desc;
```



Data export/import.

Use EMR for backup and restore
to Amazon S3.



Data export/import.

```
CREATE EXTERNAL TABLE orders_s3_new_export ( order_id
string, customer_id string, order_date int, total
double )
PARTITIONED BY (year string, month string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://export_bucket';
```

```
INSERT OVERWRITE TABLE
orders_s3_new_export
PARTITION (year='2012', month='01')
SELECT * from orders_ddb_2012_01;
```



Integrate live and archive data

Run queries across external Hive tables on S3 and DynamoDB.

Live & archive. Metadata & big objects.

In summary...



DynamoDB

Predictable performance

Provisioned throughput

Libraries & mappers

In summary...

DynamoDB

Predictable performance

Provisioned throughput

Libraries & mappers

Data modeling

Tables & items

Read & write patterns

Time series data

In summary...



DynamoDB

Predictable performance
Provisioned throughput
Libraries & mappers



Partitioning

Automatic partitioning
Hot and cold data
Size/throughput ratio



Data modeling

Tables & items
Read & write patterns
Time series data

In summary...



DynamoDB

Predictable performance
Provisioned throughput
Libraries & mappers



Partitioning

Automatic partitioning
Hot and cold data
Size/throughput ratio



Data modeling

Tables & items
Read & write patterns
Time series data



Analytics

Elastic MapReduce
Hive queries
Backup & restore



DynamoDB free tier

5 writes, 10 consistent reads per second
100Mb of storage



aws.amazon.com/dynamodb

aws.amazon.com/documentation/dynamodb



BEST PRACTICE + SAMPLE CODE

Thank you!



Q & A

matthew@amazon.com

@mza