COMSM0010 Cloud Computing

Lecture 18

Cloud Databases

Dave Cliff

Department of Computer Science
University of Bristol

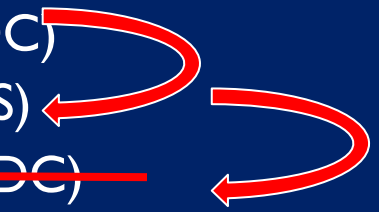
csdtc@bristol.ac.uk

The old schedule for the Big Data stuff…

- L13 Big Data I: Google's Core Tech (DC)
- L15 Big Data II: The Hadoop Ecosystem (DC)
- L16 Big Data III: Spark (DC)
- L17 Big Data IV: Cloud Databases (DC)
- L18: Big Data V: Graph Databases (DS)
- L19: Big Data VI: Stream Processing (DC)
- L20: Cloud Security (DC)

The revised schedule for the Big Data stuff…

- L13 Big Data I: Google's Core Tech (DC)
- ~~L15 Big Data II: The Hadoop Ecosystem~~ (DC) – slides + video on Blackboard
- ~~L16 Big Data III: Spark (DC)~~
- L17 Big Data IV: Cloud Databases (DC)
- L18: Big Data V: Graph Databases (DS)
- ~~L19: Big Data VI: Stream Processing (DC)~~
- L20: Cloud Security (DC)



The revised schedule for the Big Data stuff…

- L13 Big Data I: Google's Core Tech (DC)
- L15 Big Data II: The Hadoop Ecosystem (DC)
- L16 [Spark: cancelled due to strike]
- L17 [Streams: cancelled due to strike]
- L18: Big Data IV: Cloud Databases (DC)
- L19: Big Data V: Graph Databases (DS)
- L20: Cloud Security (DC)

What's coming

Setting the scene

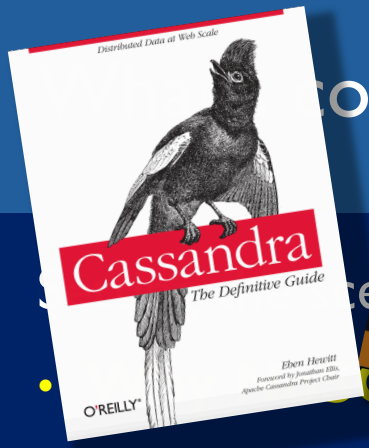
- **Why NoSQL:** scale up vs scale out

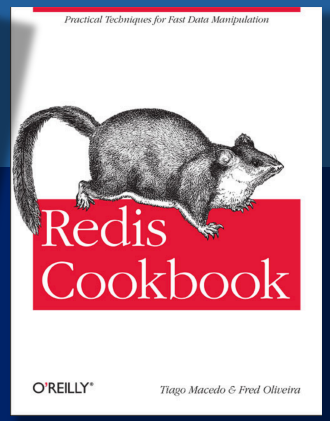
Four main classes of NoSQL DB, with popular examples

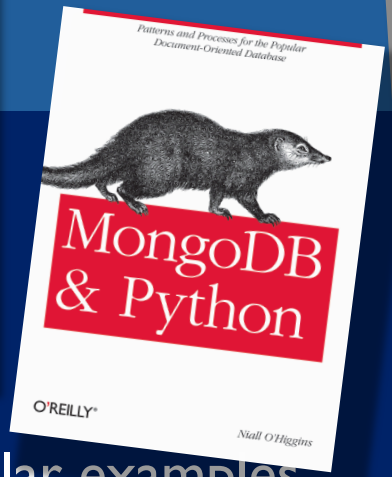
- **Key-Value DBs:** DynamoDB; Memcached; Redis; Riak
- **Document DBs**: MongoDB; CouchDB
- **Column-Family DBs:** HBase; Cassandra
- **Graph DBs:** Giraph; Neo4j; Pregel **see next lecture**

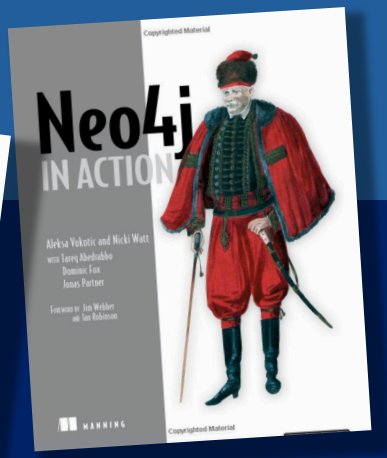
Choosing a NoSQL DB: which one? (And why stop at one?)

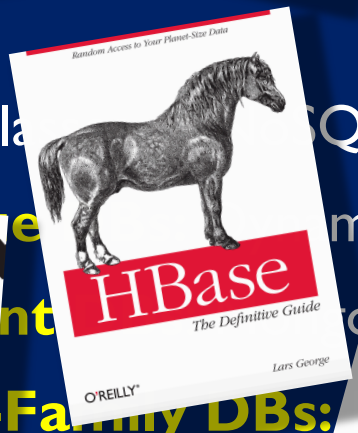
(Briefly) **NewSQL:** the future is the past

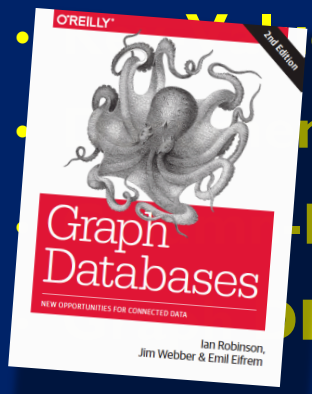


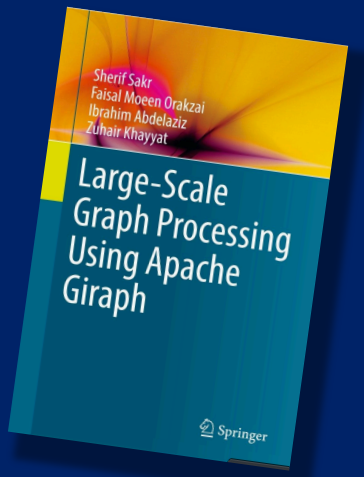










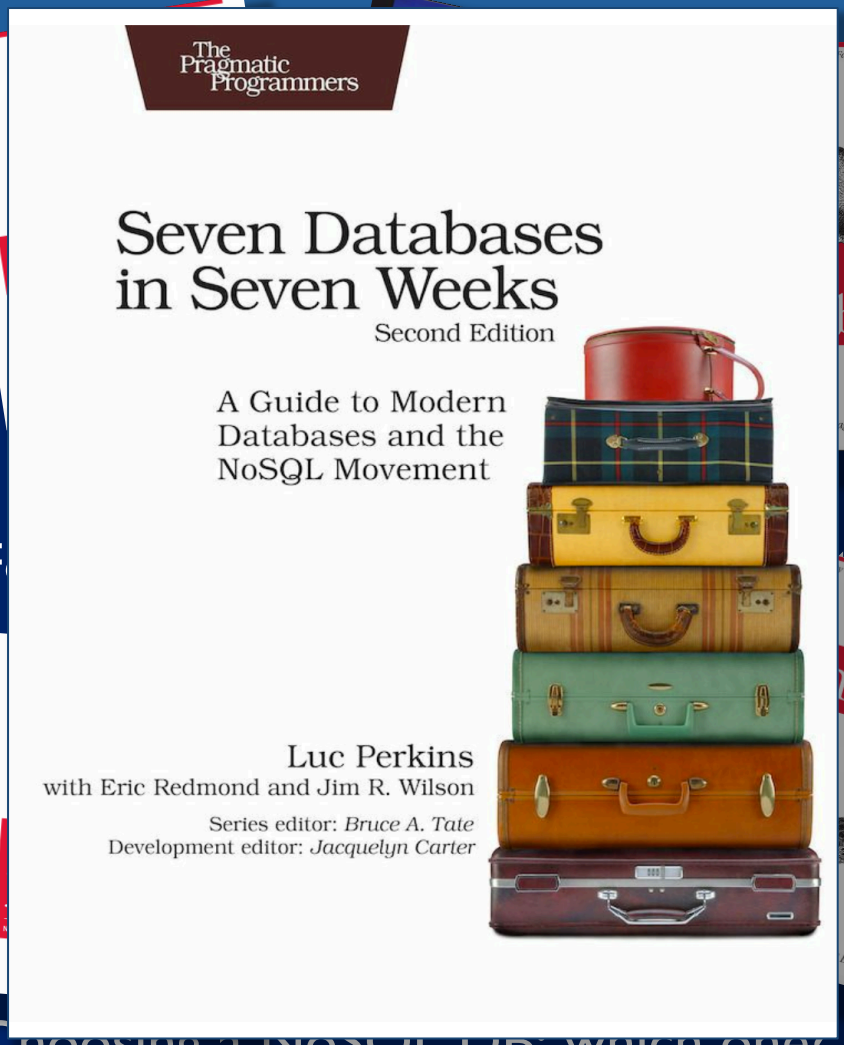


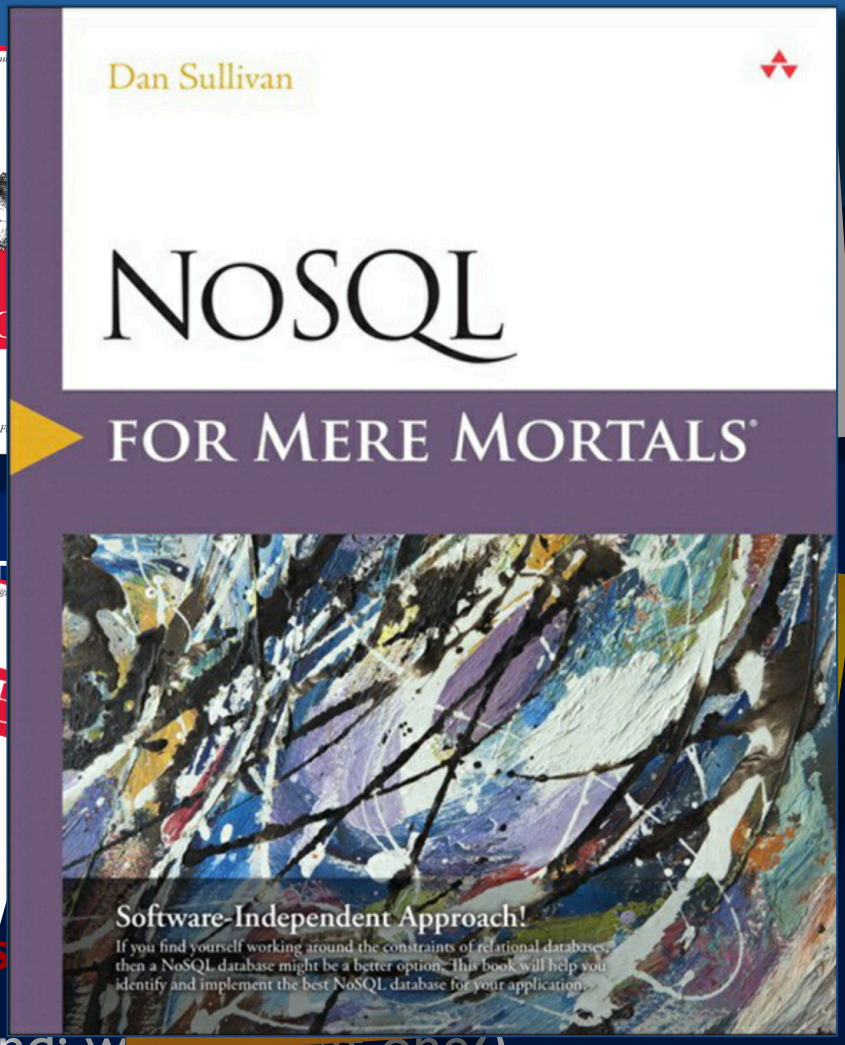
Four main cla SQL ith popular examples

- Value
- nt
- -Family DBs: ndra
- Bs: Giraph; Neo4j, see n

Choosing a NoSQL DB: which one? (And: why stop at one?)

(Briefly) **NewSQL:** the future is the past





Choosing a NoSQL DB: which one? (And: why stop at one?)

(Briefly) **NewSQL:** the future is the past

Back in Lecture 1…

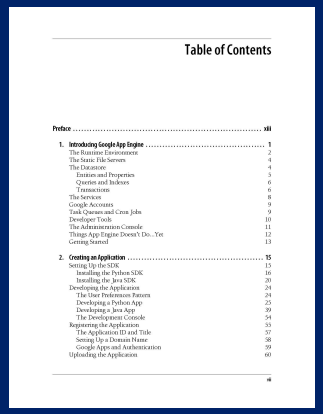


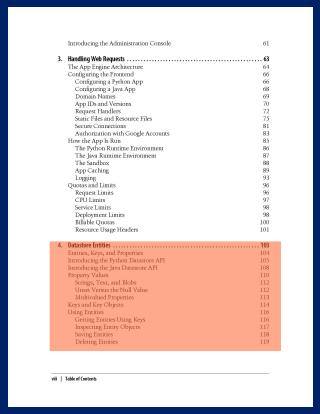
Table of Contents

Preface xiii

1. Introducing Google App Engine 1
The Runtime Environment 2
The Static File Servers 4
The Datastore 4
Entities and Properties 5
Queries and Indexes 6
Transactions 6
The Services 8
Google Accounts 9
Task Queues and Cron Jobs 9
Developer Tools 10
The Administration Console 11
Things App Engine Doesn't Do...Yet 12
Getting Started 13

2. Creating an Application 15
Setting Up the SDK 15
Installing the Python SDK 16
Installing the Java SDK 20
Developing the Application 24
The User Preferences Pattern 24
Developing a Python App 25
Developing a Java App 39
The Development Console 54
Registering the Application 55
The Application ID and Title 57
Setting Up a Domain Name 58
Google Apps and Authentication 59
Uploading the Application 60

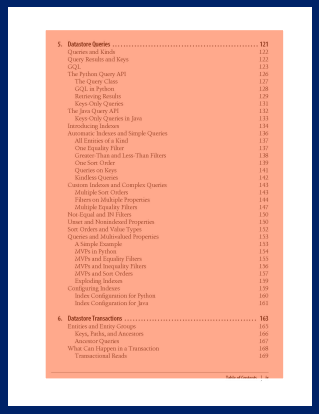
xii


Introducing the Administration Console 61

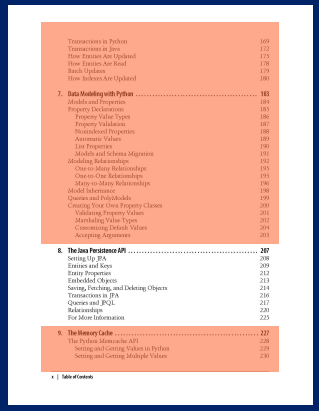
3. Handling Web Requests 63
The App Engine Architecture 64
Configuring the Frontend 66
Configuring a Python App 66
Configuring a Java App 68
Domain Names 69
App IDs and Versions 70
Request Handlers 72
Static Files and Resource Files 75
Secure Connections 81
Authorization with Google Accounts 83
How the App Is Run 85
The Python Runtime Environment 86
The Java Runtime Environment 87
The Sandbox 88
App Caching 89
Logging 93
Quotas and Limits 96
Request Limits 96
CPU Limits 97
Service Limits 98
Deployment Limits 98
Billable Quotas 100
Resource Usage Headers 101

4. Datastore Entities 103
Entities, Keys, and Properties 104
Introducing the Python Datastore API 105
Introducing the Java Datastore API 108
Property Values 110
Strings, Text, and Blobs 112
Unset Versus the Null Value 112
Multivalued Properties 113
Keys and Key Objects 114
Using Entities 116
Getting Entities Using Keys 116
Inspecting Entity Objects 117
Saving Entities 118
Deleting Entities 119

viii | Table of Contents


5. Datastore Queries 121
Queries and Kinds 122
Query Results and Keys 122
GQL 123
The Python Query API 126
The Query Class 127
GQL in Python 128
Retrieving Results 129
Keys-Only Queries 131
The Java Query API 132
Keys-Only Queries in Java 133
Introducing Indexes 134
Automatic Indexes and Simple Queries 136
All Entities of a Kind 137
One Equality Filter 137
Greater Than and Less Than Filters 138
One Sort Order 139
Queries on Keys 141
Kindless Queries 142
Custom Indexes and Complex Queries 143
Multiple Sort Orders 143
Filters on Multiple Properties 144
Multiple Equality Filters 147
Not-Equal and IN Filters 150
Unset and Nonindexed Properties 150
Sort Orders and Value Types 152
Queries and Multivalued Properties 153
A Sample Example 153
AND's in Python 154
AND's and Equality Filters 155
AND's and Inequality Filters 156
AND's and Sort Orders 157
Exploding Indexes 159
Configuring Indexes 159
Index Configuration for Python 160
Index Configuration for Java 161

6. Datastore Transactions 163
Entities and Entity Groups 165
Keys, Paths, and Ancestors 166
Ancestor Queries 167
What Can Happen in a Transaction 168
Transactional Reads 169

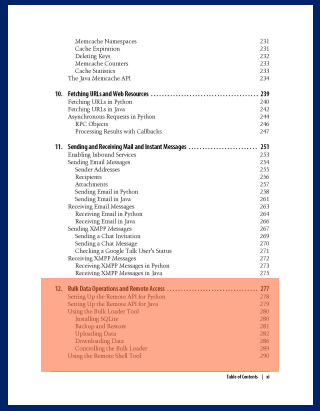

Transactions in Python 169
Transactions in Java 172
How Entities Are Updated 173
How Entities Are Read 178
Batch Updates 179
How Indexes Are Updated 180

7. Data Modeling with Python 183
Models and Properties 184
Property Declarations 185
Property Value Types 186
Property Validation 187
Nonindexed Properties 188
Automatic Values 189
List Properties 190
Models and Schema Migration 191
Modeling Relationships 192
One-to-Many Relationships 193
One-to-One Relationships 193
Many-to-Many Relationships 196
Model Inheritance 198
Queries and Polymodels 199
Creating Your Own Property Classes 200
Validating Property Values 201
Marshaling Value Types 202
Customizing Default Values 204
Accepting Arguments 205

8. The Java Persistence API 207
Setting Up JPA 208
Entities and Keys 209
Entity Properties 212
Embedded Objects 213
Saving, Fetching, and Deleting Objects 214
Transactions in JPA 216
Queries and JPQL 217
Relationships 220
For More Information 223

9. The Memory Cache 227
The Python Memcache API 228
Setting and Getting Values in Python 229
Setting and Getting Multiple Values 230

x | Table of Contents

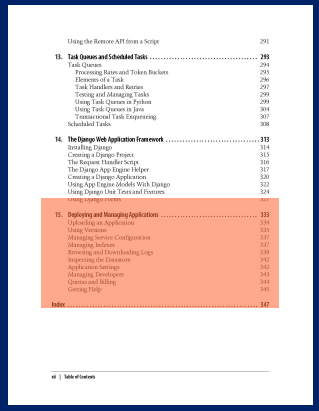

Memcache Namespaces 231
Cache Expiration 231
Deleting Keys 232
Memcache Counters 233
Cache Statistics 233
The Java Memcache API 234

10. Fetching URLs and Web Resources 239
Fetching URLs in Python 240
Fetching URLs in Java 242
Asynchronous Requests in Python 244
RPC Objects 246
Processing Results with Callbacks 247

11. Sending and Receiving Mail and Instant Messages 251
Enabling Inbound Services 253
Sending Email Messages 254
Sender Addresses 255
Recipients 256
Attachments 257
Sending Email in Python 258
Sending Email in Java 261
Receiving Email Messages 263
Receiving Email in Python 264
Receiving Email in Java 266
Sending XMPP Messages 267
Sending a Chat Invitation 269
Sending a Chat Message 270
Checking a Google Talk User's Status 271
Receiving XMPP Messages 272
Receiving XMPP Messages in Python 273
Receiving XMPP Messages in Java 273

12. Bulk Data Operations and Remote Access 277
Setting Up the Remote API for Python 278
Setting Up the Remote API for Java 279
Using the Bulk Loader Tool 280
Installing SQLite 280
Backup and Restore 281
Uploading Data 282
Downloading Data 286
Controlling the Bulk Loader 289
Using the Remote Shell Tool 290

Table of Contents | xi


Using the Remote API from a Script 291

13. Task Queues and Scheduled Tasks 293
Task Queues 294
Processing Rates and Token Buckets 293
Elements of a Task 296
Task Handlers and Retries 297
Testing and Managing Tasks 299
Using Task Queues in Python 299
Using Task Queues in Java 304
Transactional Task Enqueueing 307
Scheduled Tasks 308

14. The Django Web Application Framework 313
Installing Django 314
Creating a Django Project 313
The Request Handler Script 316
The Django App Engine Helper 317
Creating a Django Application 320
Using App Engine Models With Django 322
Using Django Unit Tests and Fixtures 324

15. Deploying and Managing Applications 333
Uploading an Application 334
Using Versions 335
Managing Service Configuration 337
Managing Indexes 337
Browsing and Downloading Logs 339
Inspecting the Datastore 342
Application Settings 342
Managing Developers 343
Quotas and Billing 344
Getting Help 345

Index 347

xii | Table of Contents

>40% of this book is all about "dataSTORES"

When dinosaurs roamed the earth

Traditional RDBMs designed for (essentially) book-keeping/accounting

- Fundamental data structure is table divided into rows and columns
- Database schema defines what is allowed in each cell of a table

Relational databases hugely popular successful from 1970s onwards

- Each table defines a relation between records (rows) and attributes (cols)
- E.g. Table A might be customer records, table B might be order records
  - Query might be "List all customers living in Bristol who purchased product P"
  - Table B has I.D.s of customers who purchased P; table A has address location
  - JOIN the two tables to get the answer
  - RDBMSs offer Atomicity, Consistency, Isolation and Durability (ACID)

When dinosaurs roamed the earth

Traditional RDBMs designed for (essentially) book-keeping/accounting

- Fundamental data structure is table divided into rows and columns
- Database schema defines what is allowed in each cell of a table

Relational databases hugely popular successful from 1970s onwards

- Each table defi (rows) and attributes (cols)
- E.g. Table A m B might be order records
  - Query might be "Li stomers living in Bristol who purchased product P"
  - Table B has I.D.s o omers who purchased P; table A has address location
  - JOIN the two tabl to get the answer
  - RDBMSs offer Atomicity, Consistency, Isolation and Durability (ACID)

Atomicity: each transaction is all-or-nothing

When dinosaurs roamed the earth

Traditional RDBMs designed for (essentially) book-keeping/accounting

- Fundamental data structure is table divided into rows and columns
- Database schema defines what is allowed in each cell of a table

Relational databases hugely popular successful from 1970s onwards

- Each table defines a rel… …nd attributes (cols)
- E.g. Table A might be c… …be order records
  - Query might be "List all custo… …ving in Bristol who purchased product P"
  - Table B has I.D.s of customers … purchased P; table A has address location
  - JOIN the two tables to get the …swer
  - RDBMSs offer Atomicity, Consistency, Isolation and Durability (ACID)

Consistency: data should be valid across the system

When dinosaurs roamed the earth

Traditional RDBMs designed for (essentially) book-keeping/accounting

- Fundamental data structure is table divided into rows and columns
- Database schema defines what is allowed in each cell of a table

Relational databases hugely popular successful from 1970s onwards

- Each table defines a relati… …attributes (cols)
- E.g. Table A might be cust… …order records
  - Query might be "List all customers living in Br… …purchased product P"
  - Table B has I.D.s of customers who purchase… …able A has address location
  - JOIN the two tables to get the answer
  - RDBMSs offer Atomicity, Consistency, Isolation and Durability (ACID)

Isolation: transactions do not affect each other

When dinosaurs roamed the earth

Traditional RDBMs designed for (essentially) book-keeping/accounting

- Fundamental data structure is table divided into rows and columns
- Database schema defines what is allowed in each cell of a table

Relational databases hugely popular successful from 1970s onwards

- Each table defines a relation betwee cols)
- E.g. Table A might be customer reco rds
  - Query might be "List all customers living in Bristol who pu product P"
  - Table B has I.D.s of customers who purchased P; table A ddress location
  - JOIN the two tables to get the answer
  - RDBMSs offer Atomicity, Consistency, Isolation and Durability (ACID)

Durability: once it's committed, data is not lost

When dinosaurs roamed the earth

Traditional RDBMs designed for (essentially) book-keeping/accounting

- Fundamental data structure is table divided into rows and columns
- Database schema defines what is allowed in each cell of a table

Relational databases hugely popular successful from 1970s onwards

- Each table defines a relation between records (rows) and attributes (cols)
- E.g. Table A might be customer records, table B might be order records
  - Query might be "List all customers living in Bristol who purchased product P"
  - Table B has I.D.s of customers who purchased P; table A has address location
  - JOIN the two tables to get the answer
  - RDBMSs offer Atomicity, Consistency, Isolation and Durability (ACID)
- But in a lot if situations you just want CRUD: Create/Read/Update/Delete

Why NoSQL

Traditional RDBMs were designed to run on a single server

As databases got bigger, people designed bigger single servers

- Costs increased nonlinearly
- Eventually moved to highly engineered & very tightly controlled clustered RDBMSs
- Costs still really not at all cheap

Even when scaled up, RDBMSs suffered from "inelasticity"

- Once scaled up, they were hard to scale back down again if/when demands reduced

Relational model just got in the way. If you want to do CRUD on clusters of cheap commodity servers, and you don't care (or can work around) lack of ACID, there are other ways of doing things… NoSQL ways…

Key-Value DBs

Key-Value DBs: Basics

Simplest of NoSQL DB types: minimal constraints; "schemaless"

Often very fast

Extensible: don't require heavy up-front investment in designing schema

Essentially similar to an *associative array*:

- array[*0*] … array[*n*] uses integer *i* as *index*: associates each *i* with a value
- Assoc.array allows any value, any type to be the index: this is the key

Keys must be unique within any one KV namespace

- Namespace may be whole DB, or one of several buckets within a DB

Value can be pretty much any digital object

- In practice, different KV DB systems impose their own constraints

Key-Value DBs: Details

KV pairs often mapped to server nodes via a hash function to balance storage/processing load over them

No standard for KV datastores; different KVs have differing constraints:

- Some offer ACID transactions; others not
- Some allow very big objects in values; others have hard limits
- Some constrain data-types of keys to be only strings/numbers, others not

Some KV systems have built-in indexing of values

- As KV pairs are added to DB, an index is compiled of unique values
- Produces a value->key LUT; index tells you all the keys for a given value

Some KV systems auto-compress: keys/values on disk are compressed

Key-Value DBs: Examples

Start simple…

Amazon SimpleDB: basics

Source: Murty

Provides:

- Reliable storage of structured textual data
- Language that allows you to store, modify, query, & retrieve data sets
- Automatic indexing of all your stored data

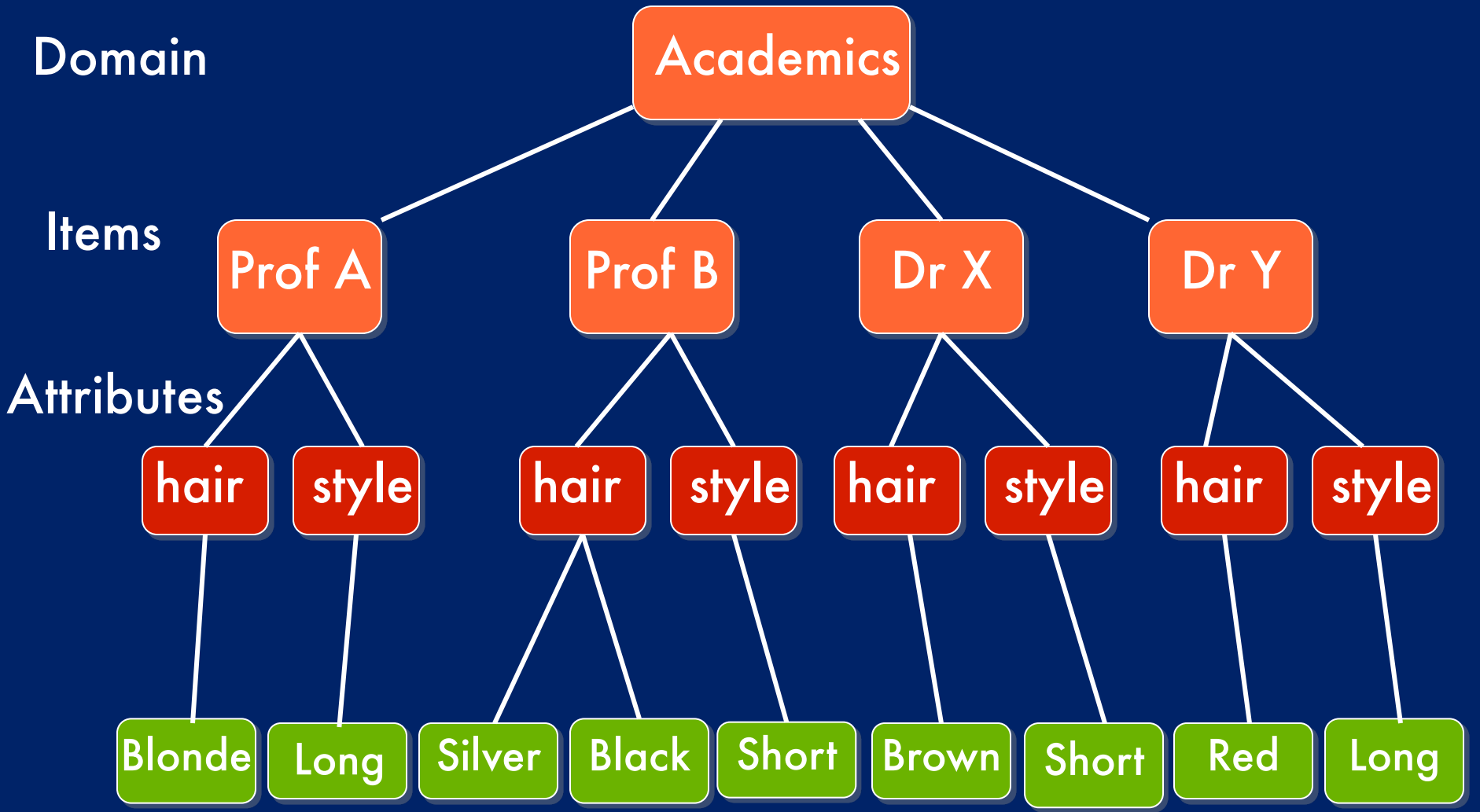
SimpleDB provides 3 main resources:

- **Domains:** highest-level container for related data items; queries only search within one domain

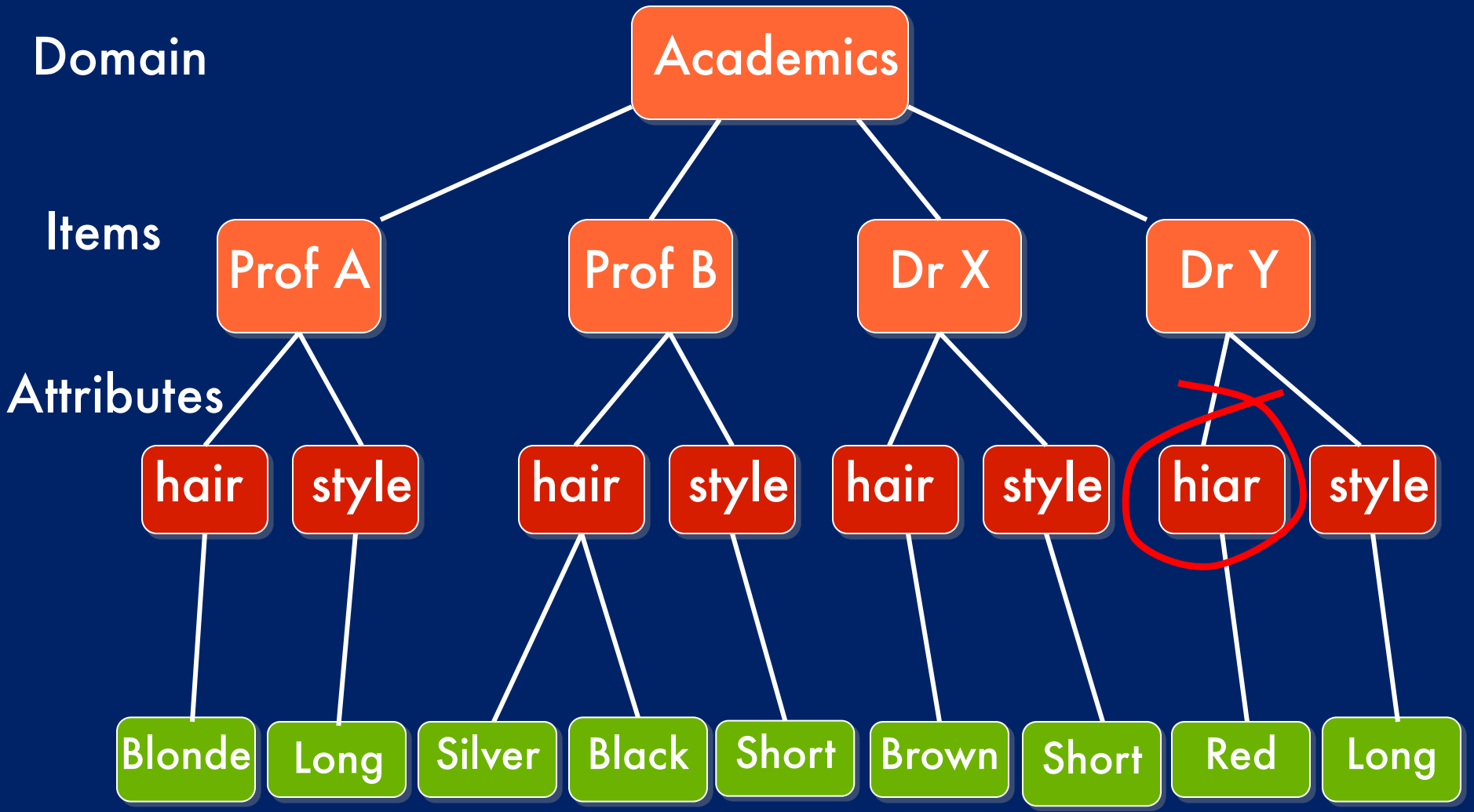
- **Items:** a named collection of attributes that represents a data object; each item has a unique name within the domain; items can be created, modified, or deleted; individual attributes within an item can be manipulated

- **Attributes:** an individual category of information within the item, with a name unique for that item; item has one or more text string values associated with the name

Amazon SimpleDB: an example hierarchy



Amazon SimpleDB: an example hierarchy



Amazon SimpleDB: not an RDBMS

Source: Murty

NOT a traditional Relational Database Management System (RDBMS)
– it is simple

Primary differences:

- SimpleDB Items are stored in a hierarchical structure, not a table
- SimpleDB attribute value max size =1Kb
- SimpleDB data is all stored as text – there are no other data-types!
- SimpleDB's query language is, um, simple in comparison to SQL et al
- SimpleDB is distributed: data consistency may suffer due to propagation delays

Amazon Sim

Source: Murty

NOT a traditional [illegible] DBMS)
– it is simple

Primary differenc

- SimpleDB Iter [illegible] table
- SimpleDB att
- SimpleDB d [illegible] ata-types!
- SimpleDB's [illegible] to SQL et al
- SimpleDB [illegible] e to propagation delays

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

SOSP 2007

GAE Datastore

Source: Sanderson (2013) Ch.1

Storing data, holding it from one request to another, is clearly useful.

Pre-cloud: a single database server would be scaled up (and up, and up…)

- …Almost always a relational database

GAE approach, the Datastore, is cloud style: scale-out, not scale-up

Datastore scale-out is automated – you don't have to worry about how it is distributed over multiple servers

But you have to sacrifice relational database Join-Query mode of working

GAE Datastore holds data as entities;

Each entity has one or more properties,

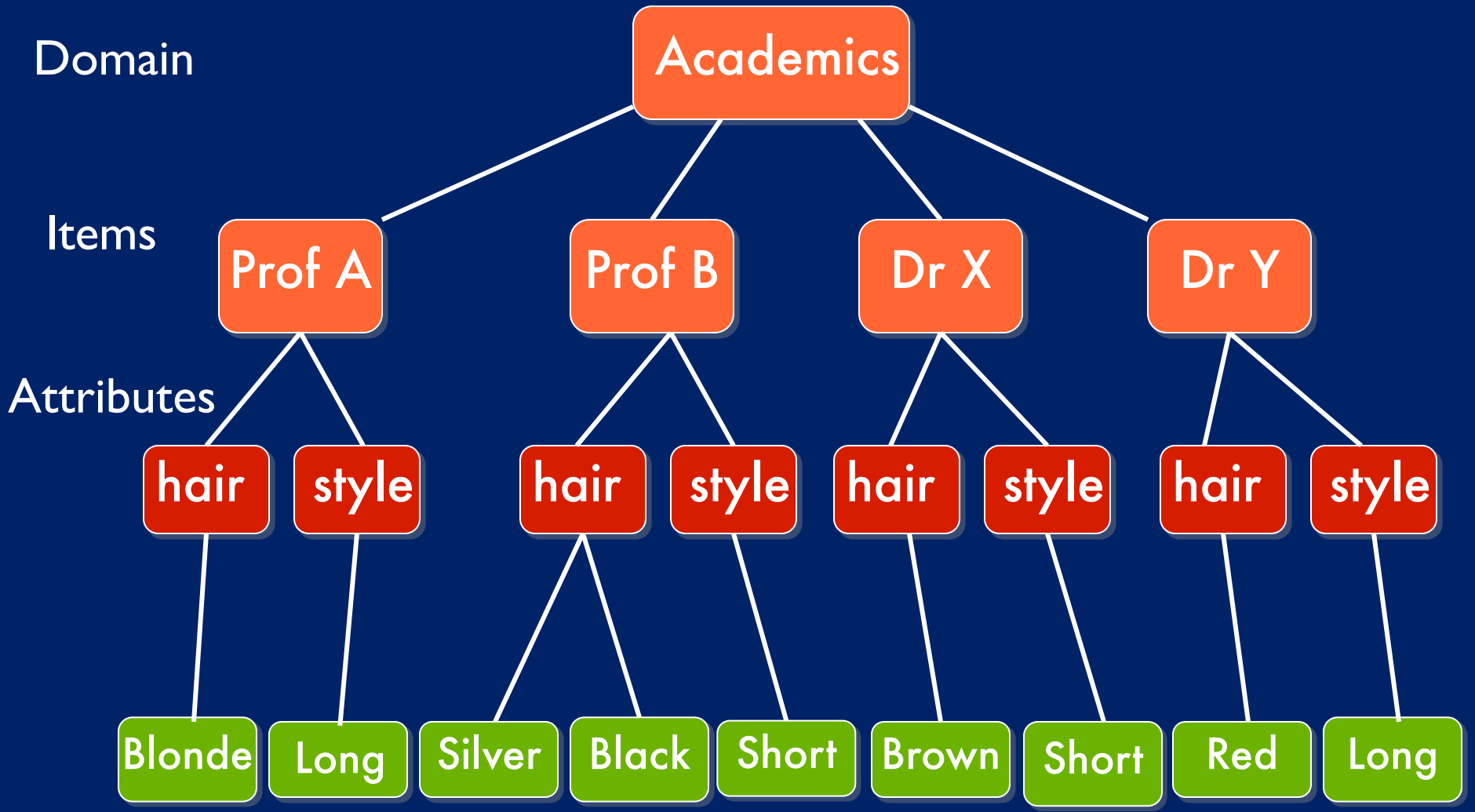
Each property has a name, and one or more values (the value is one of several primitive types);

Entity belongs to a kind (like a category or class) + a unique key (within kind)

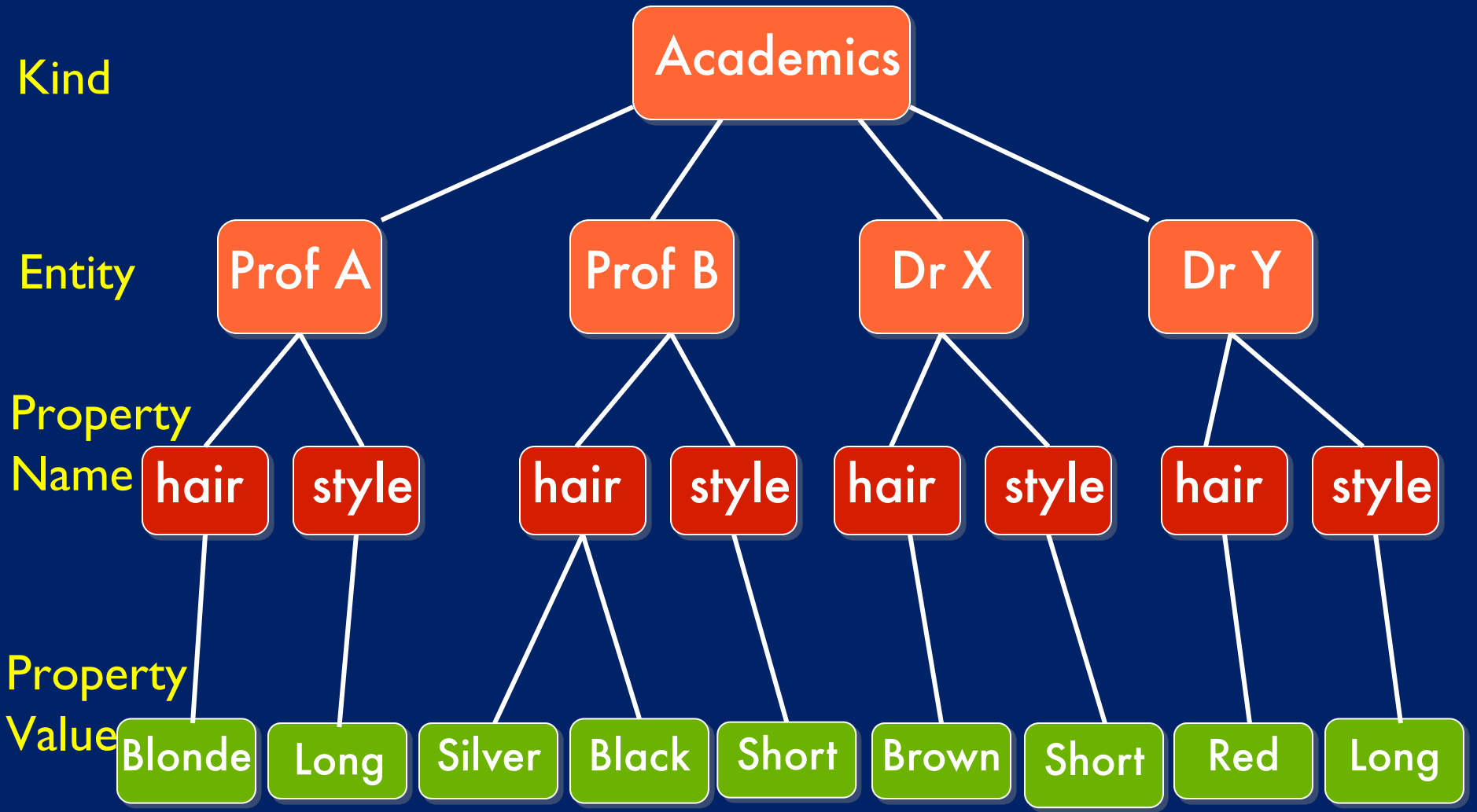
GAE Datastore

Let's go back to that AWS SimpleDB example…

Amazon SimpleDB: an example hierarchy



GAE Datastore



GAE Datastore: Queries and (pre-)Indexes

Source: Sanderson (2013) Ch1.

Datastore queries return entities of a single kind, or just their keys

GQL queries can filter entities on the basis of their property values

GQL query results can be returned in sorted order, of property-values

GQL can filter/sort on keys too

Big idea: query pre-indexing

For every allowable query, the datastore pre-builds an index

- GAE SDK watches which queries are used in development & testing
- When you upload your app, the datastore makes indexes for every query that the SDK saw during developer's testing (this list can be edited)
- Means resolving/answering queries is very fast
- But adding/updating entities can create a lot of book-keeping

GAE Datastore

(As you can imagine, we are skipping over some details here)

App instances in the RE access the datastore as a service, via an API.

The datastore self-scales, but the app doesn't need to know how.

Other self-scaling GAE services

Source: Sanderson (2013) Ch1.

Memcache

- Short-term key-value storage service
- Uses RAM not disk; hence much faster than datastore
- Unlike datastore, it is not persistent… server down => data-loss
- Best used as a cache – hold recent results from frequent actions

Blobstore

- Storage for large items, like images, video, or other big files

URL Fetch

- HTTP request to another server on the Internet

Back to AWS…

Amazon Sim

Source: Murty

NOT a traditional [illegible] DBMS)
– it is simple

Primary differenc

- SimpleDB Iter [illegible] table
- SimpleDB att
- SimpleDB d [illegible] ata-types!
- SimpleDB's [illegible] to SQL et al
- SimpleDB [illegible] e to propagation delays

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

SOSP 2007

Amazon Sim

NOT a traditional

Source: Mur

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels
Amazon.com

ABSTRACT
Reliability at massive

Contact Sales | Support | English | My Account | Sign In to the Console

aws | re:Invent 2018 | Products | Solutions | Pricing | Learn | Partner Network | AWS Marketplace | Explore More

Amazon DynamoDB | Overview | Features | Pricing | Getting Started | Migrations | Resources | FAQs

Amazon DynamoDB

Fast and flexible NoSQL database service for any scale

Get started with Amazon DynamoDB

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multiregion, multimaster database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB can handle more than 10 trillion requests per day and support peaks of more than 20 million requests per second.

Lyft, Airbnb, and Redfin as well as enterprises such as

Amazon Sim

NOT a traditional

Source: Mu

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive face at

aws

Contact Sales | Support | English | My Account | Sign In to the Console

re:Invent 2018 | Products | Solutions | Pricing | Learn | Partner Network | AWS Marketplace | Explore More

Amazon DynamoDB | Overview | Features | Pricing | Getting Started | Migrations | Resources | FAQs

Amazon DynamoDB

Amazon DynamoDB is a key-value and document database th

any scale. It's a fully managed, multiregion, multimaster datab

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multiregion, multimaster database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB can handle more than 10 trillion requests per day and support peaks of more than 20 million requests per second.

Lyft, Airbnb, and Redfin as well as enterprises such as

Document DBs

Document DBs: Basics

Source: Sullivan, 2015; Ch.6.

- Document DB manage more complex data structures than KV datastores
- Like KVs (and unlike RDBMSs) DocDBs don't require you to define a common structure for all records in the data store
- Unlike KVs (but like RDBMSs), DocDBs allow you to query and filter collections of documents, as you would rows in a RDB table

- NB they are NOT stores of electronic documents (MSWord files, Spreadsheets, etc) – these can be stored in document management systems but that is not what is mean in the context of NoSQL DBs

- In NoSQL, a "document" is a structured object: it contains information not only about the data being stored, but also how that data is structured.
- So a document in a DocDb is at root a set of key-value pairs, but some of the values can themselves be sets of key-value pairs (and on and on)
- Most commonly, documents are defined in either JSON or XML. …

Document DBs: Basics

Source: Sullivan, 2015; Ch.6.

- Document DB manage more complex data structures than KV datastores
- Like KVs (and unlike RDBMSs) DocDBs don't require you to define a common structure for all records in the data store
- Unlike KVs (but like RDBMSs), DocDBs allow you to query and filter collections of documents, as you would rows in a RDB table

- NB they are NOT stores of electronic documents (MSWord files, Spreadsheets etc) – these can be stored in document management systems but that is not what is mean in the context of NoSQL DBs

- In NoSQL, a "document" is a struct … ormation not only about the data being stored … structured.
- So a document in a DocDb is at root a set of key-va … pairs, but some of the values can themselves be sets of key-value pairs (and on and on)
- Most commonly, documents are defined in either JSON or XML. …

JSON: JavaScript Object Notation

Document DBs: Example Documents

Source: Sullivan, 2015; Ch.6.

JSON

```
{
     "customer_id":187693,
    "name": "Kiera Brown",
    "address" : {
        "street" : "1232 Sandy Blvd.",
        "city" :  "Vancouver",
        "state" :  "Washington",
        "zip" :  "99121"
       },
    "first_order" : "01/15/2013",
    "last_order" : " 06/27/2014"
}
```

XML

```
<customer_record>
<customer_id>187693</customer_id>
 <name>"Kiera Brown"</name>
 <address>
    <street>"1232 Sandy Blvd."</street>
    <city>"Vancouver"</city>
    <state>"Washington"</state>
    <zip>"99121"</zip>
 </address>
 <first_order>"01/15/2013"</first_order>
 <last_order>"06/27/2014"</last_order>
</customer_record>
```

Document DBs: Example Documents

Source: Sullivan, 2015; Ch.6.

JSON

```
{
     "customer_id":187693,
    "name": "Kiera Brown",
    "address" : {
        "street" : "1232 Sandy Blvd.",
        "city" :  "Vancouver",
        "state" :  "Washington",
        "zip" :  "99121"
       },
    "first_order" : "01/15/2013",
    "last_order" : " 06/27/2014"
}
```

XML

```
<customer_record>
<customer_id>187693</customer_id>
 <name>"Kiera Brown"</name>
 <address>
    <street>"1232 Sandy Blvd."</street>
    <city>"Vancouver"</city>
    <state>"Washington"</state>
    <zip>"99121"</zip>
 </address>
 <first_order>"01/15/2013"</first_order>
 <last_order>"06/27/2014"</last_order>
</customer_record>
```

Document DBs: Examples

Mongo (from humongous) widely cited as most popular NoSQL DB

- Developed as PaaS by 10Gen Inc from 2007; free/open-source in 2009
- 2013 10Gen renamed as MongoDB Inc; 2017 MongoDB IPO
- Now offered via proprietary server-side public license (cf GPL)
- MongoDB users include Foursquare, IBM, Gap, Uber, & Urban Outfitters
- MongoDB has had some security issues: >10,000 installations ransomed

Couch (from Cluster Of Unreliable Commodity Hardware)

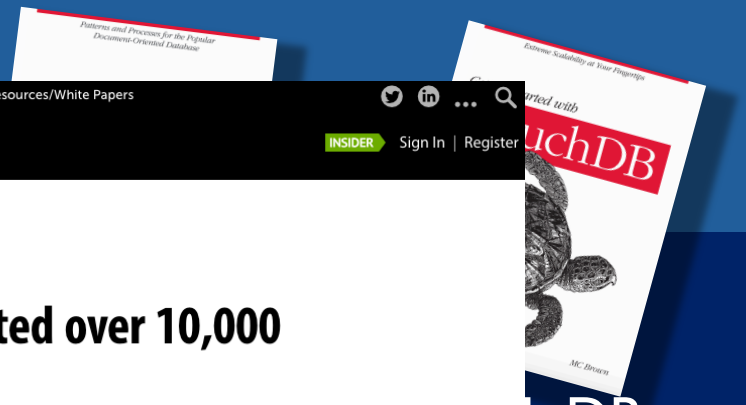
- Apache top-level project; written in Erlang; documents are in JSON
- Query language is JavaScript->MapReduce; ACID semantics; nicely offline.
- Users include BBC, CANAL+

Documen

Mongo (fro
- Developed
- 2013 10G
- Now offer
- MongoDB
- MongoDB

Couch (fro
- Apache to
- Query lang
- Users incl

L DB
e in 2009
)
n Outfitters
ransomed
JSON
icely offline.



NEWS

Ransomware groups have deleted over 10,000 MongoDB databases

Five groups of attackers are competing to delete as many publicly accessible MongoDB databases as possible

By Lucian Constantin

Romania Correspondent, IDG News Service | JAN 6, 2017 10:02 AM PT


Gerd Altmann (CC0)

Groups of attackers have adopted a new tactic that involves deleting publicly exposed MongoDB databases and asking for money to restore

MORE LIKE THIS


Attackers start wiping data from CouchDB and Hadoop databases


After MongoDB attack, ransomware groups hit exposed Elasticsearch clusters


Ransomware became one of the top threats to enterprises this year


VIDEO
Mingis on Tech: Data breaches in a world of 'surveillance capitalism'

Column-Family DBs
& Columnar DBs

Column-Family DBs: Basics

Source: Sullivan, 2015; Ch.9.

- Column-family DBs, AKA Wide-Column DBs AKA Columnar DBs
- Quite definitely for ”big data”: the technical term is VLDB, Very Large DB – say billions of rows and/or tens of thousands of columns
- RDBMSs could potentially scale to VLDB working on a few very large servers, but cost increases are likely to be prohibitive/insane.
- KV DBs scale nicely but lack features for organizing many columns for keeping frequently-used data together.
- DocDBs can also in principle scale nicely, but they lack powerful query languages.

- 2006: Google BigTable paper. Birth of “column family” DBs: very large scale NoSQL for Google’s web indexing, Google Earth, and Google Finance businesses.
- Led to HBase & Cassandra (Facebook) as open-source derivatives

Column-Family DBs

- "Column family" comes from the fact that groups of related columns, e.g. those that are frequently used together, can be gathered into groups – called families. Columns within a family are kept together on disk, to reduce access times.
- Data-modeler defines the column *families* prior to implementing the database, but developers can subsequently add columns to a family. So families are a bit like relational tables while individual columns are more like key-value pairs.
- This means that the designer of a database specifies only a course-grained structure, the set of families. Column families are analogous to keys in KV DBs – similar constraints wrt uniqueness of names in a namespace
- Like document databases, ColFam DBs do not require column entries in all rows.
- Like RDBMSs, ColFam DBs rely on unique identifiers for each row of data – known as "row keys" in ColFam and as "primary keys" in RDBMs – in both cases, auto-indexed for rapid retrieval.

Column-Family DBs: Examples

Cassandra is in top-10 most popular DB list

- Users: Apple; AppScale; BlackRock; CERN; Cisco; Digg; Netflix; Nutanix; Rackspace; Reddit; Soundcloud; Uber and more.
- Not tied to Hadoop backend (but does work with Hadoop)

HBase part of the Hadoop Ecosystem

- Users: Adobe; AirBnB; Amadeus IT; Bloomberg; Facebook; Netflix; Pinterest; Salesforce; Sears; Spotify; Yahoo!

Summary: choices

Choosing a NoSQL DB

Source: Perkins *et al.*, 2018; Ch.9.

Relational: 2D tables with rows & columns, strictly enforced data-types

Good for:

- When layout of data is known in advance but exact use/queries are not

Less Good for:

- When data is highly variable and/or deeply hierarchical

Choosing a NoSQL DB

Source: Perkins *et al.*, 2018; Ch.9.

Key-Value: maps simple keys to values

Good for:

- Horizontal scalability and/or speed
- Situations where data are largely independent
- CRUD (and not much else)

Less Good for:

- When you want to perform nontrivial queries on the data

Choosing a NoSQL DB

Source: Perkins *et al.*, 2018; Ch.9.

Document: any number of fields per object; objects may be nested; deployments often distributed (easy to shard & replicate over n>1 servers)

Good for:

- Highly variable domains/applications where you don't know in advance what your data will look like
- Situations where storing multiple redundant copies of data is not a problem (i.e. *denormalized*): each doc should have most/all necessary data on-board

Less Good for:

- Situations where data needs to be normalized (reduce/eliminate copies) and where join queries are natural mode of working

Choosing a NoSQL DB

Source: Perkins *et al.*, 2018; Ch.9.

Columnar: Sort of hybrid RDBMS/KV

Good for:

- "Big Data" where scale-out capability is key
- When you need data compression and versioning
- When you have some idea of what queries you will need to run

Less Good for:

- When you don't know how the data will be queried/used

Choosing a NoSQL DB

Source: Perkins *et al.*, 2018; Ch.9.

Graph: 2D tables with rows & columns, strictly enforced data-types

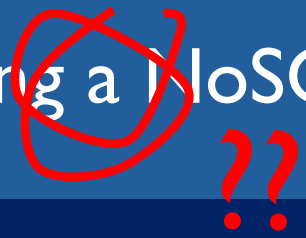
Good for:

- Applications with *networks of relationships* at their core
- Social networks; bioinformatics: social med, genomics, & proteomics
- Reasoning about the graph on a single (large?) DB server/node

Less good for:

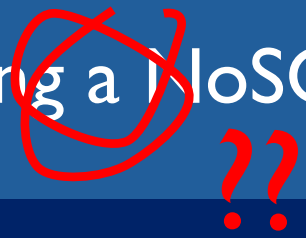
- Large-scale situations where partitioning across nodes is necessary

Choosing a NoSQL DB



Source: Perkins *et al.*, 2015; Ch.8.

Choosing a NoSQL DB



Source: Perkins *et al.*, 2015; Ch.8.

Maybe asking which *one* DB tech to use is the wrong question?

Many systems now built with multiple different DBs each playing a role

"Polyglot persistence model" (see https://martinfowler.com/bliki/PolyglotPersistence.html)

Simple example: using a KV store like Redis as a cache for queries against a relatively slower RDBMS.

More generally

- Move away from a single central DB as source of whole-system "truth"
- Instead have >1 database, with multiple types of database as necessary

Choosing a NoSQL DB

Source: Perkins *et al.*, 2015; Ch.8.

Maybe asking which *one* DB tech to use is the wrong question?

Many systems now built with multiple different DBs each playing a role

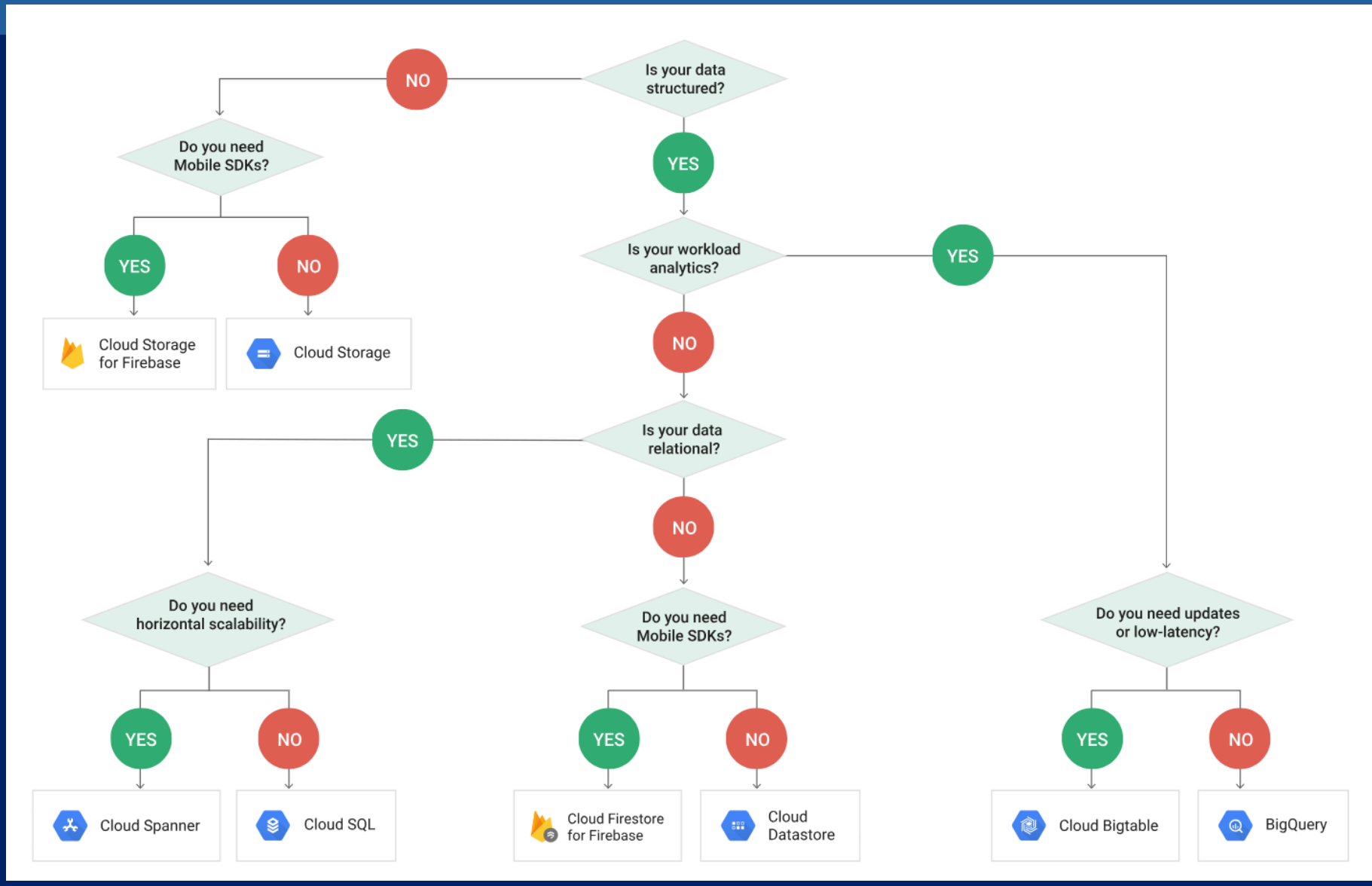
"Polyglot persistence model" (see https://martinfowler.com/bliki/PolyglotPersistence.html)

Sim nst a
rela

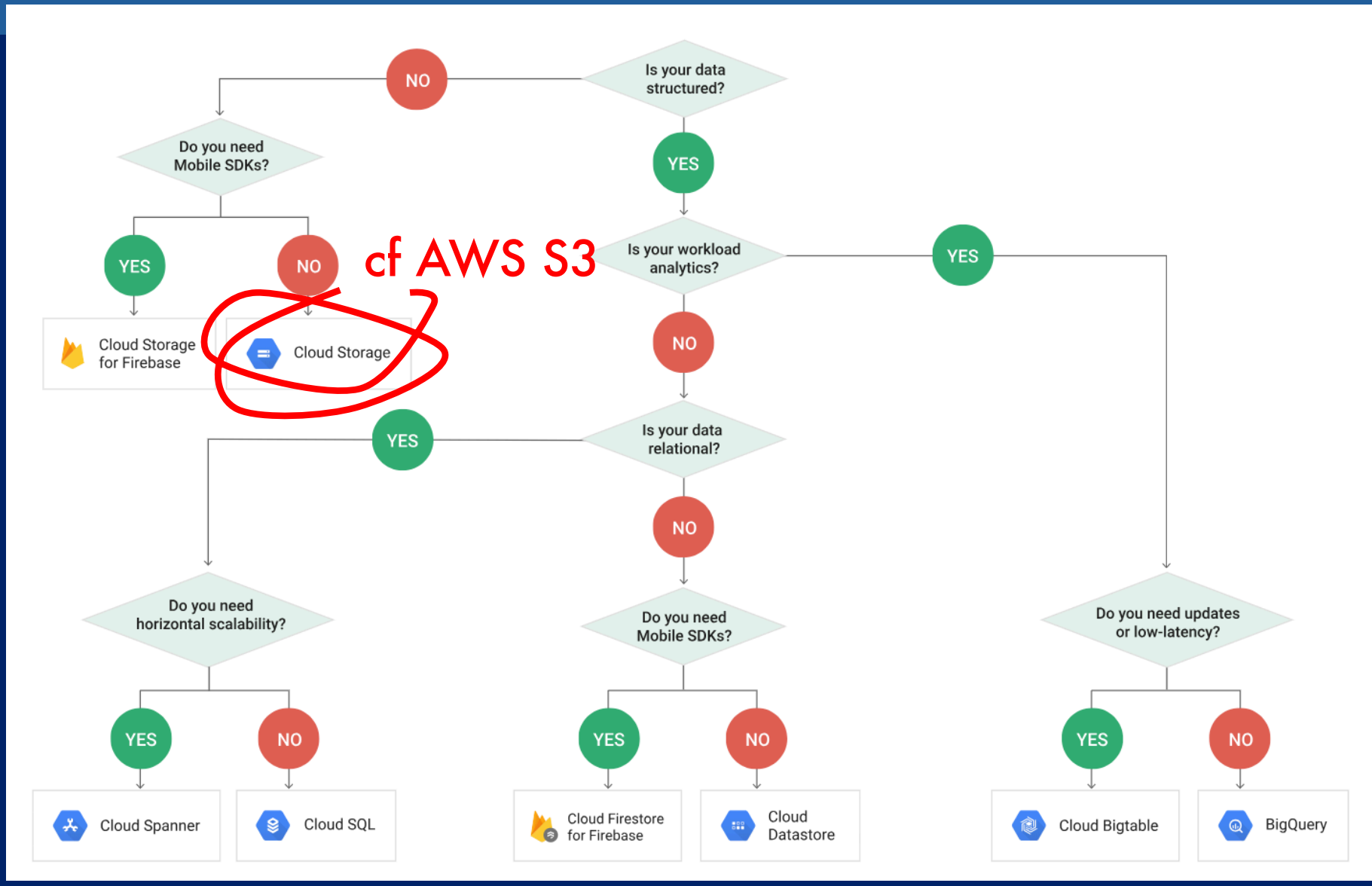
Mor

- M th"
- In ry

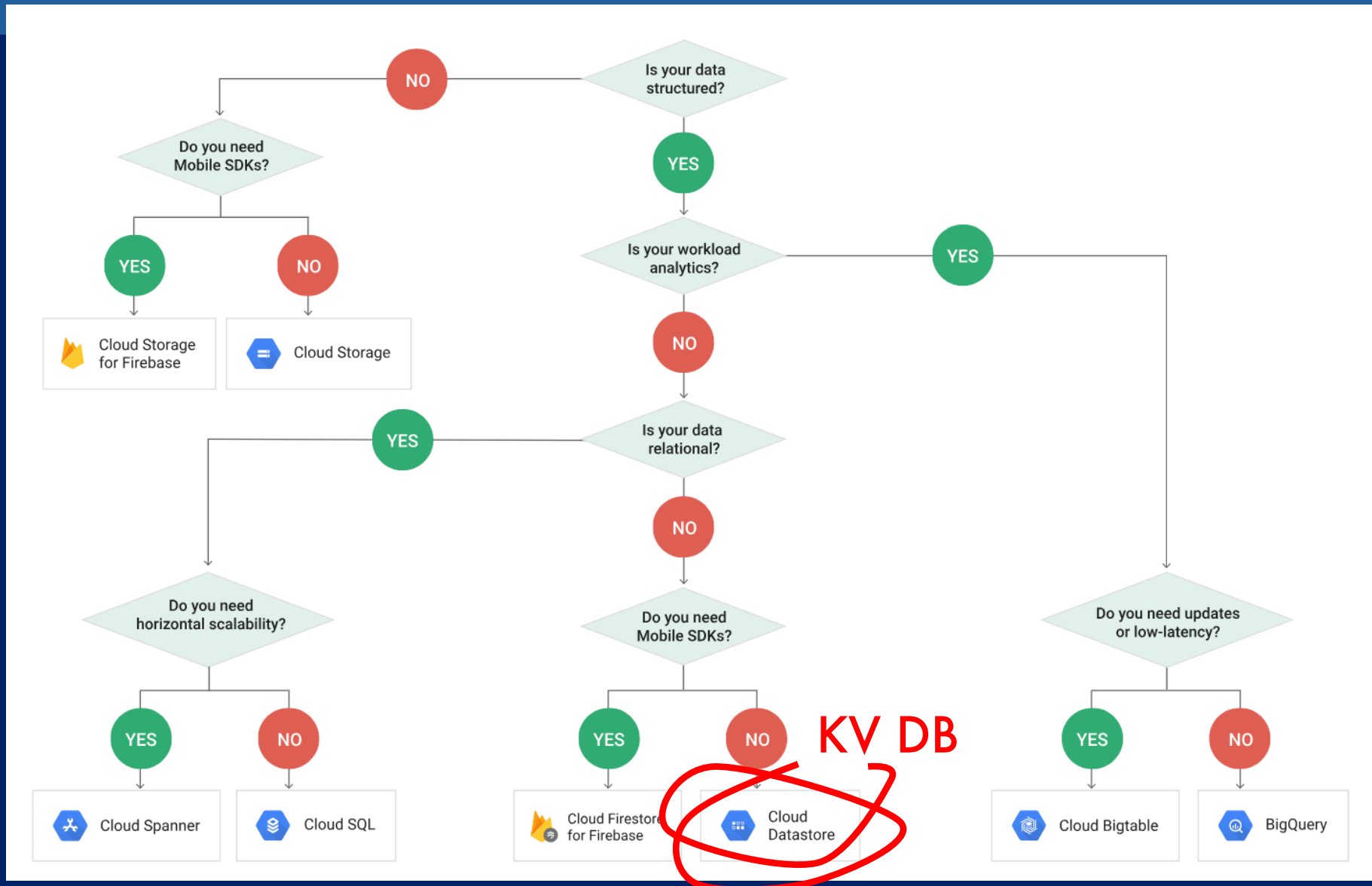
https://cloud.google.com/storage-options/



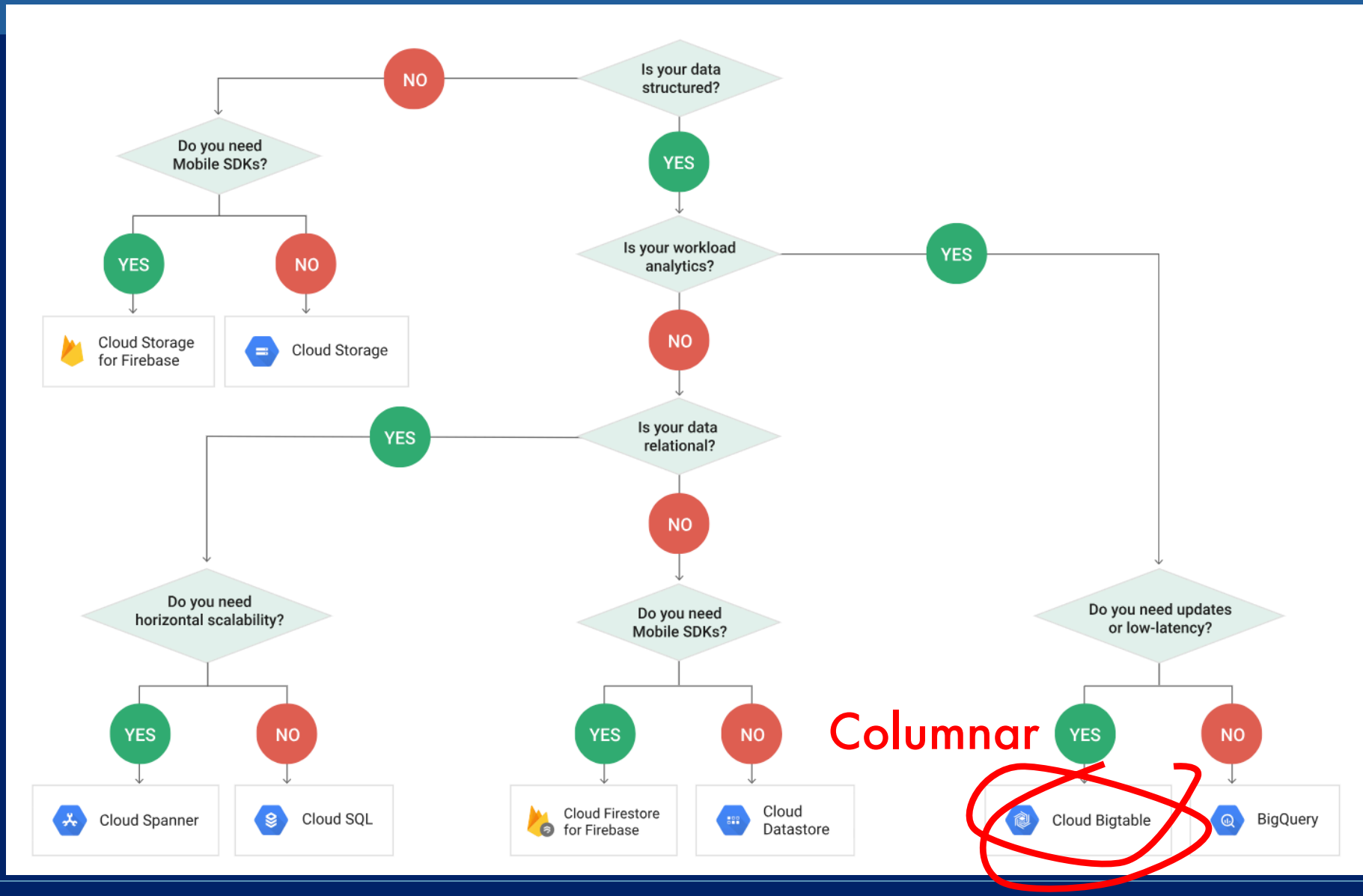
https://cloud.google.com/storage-options/



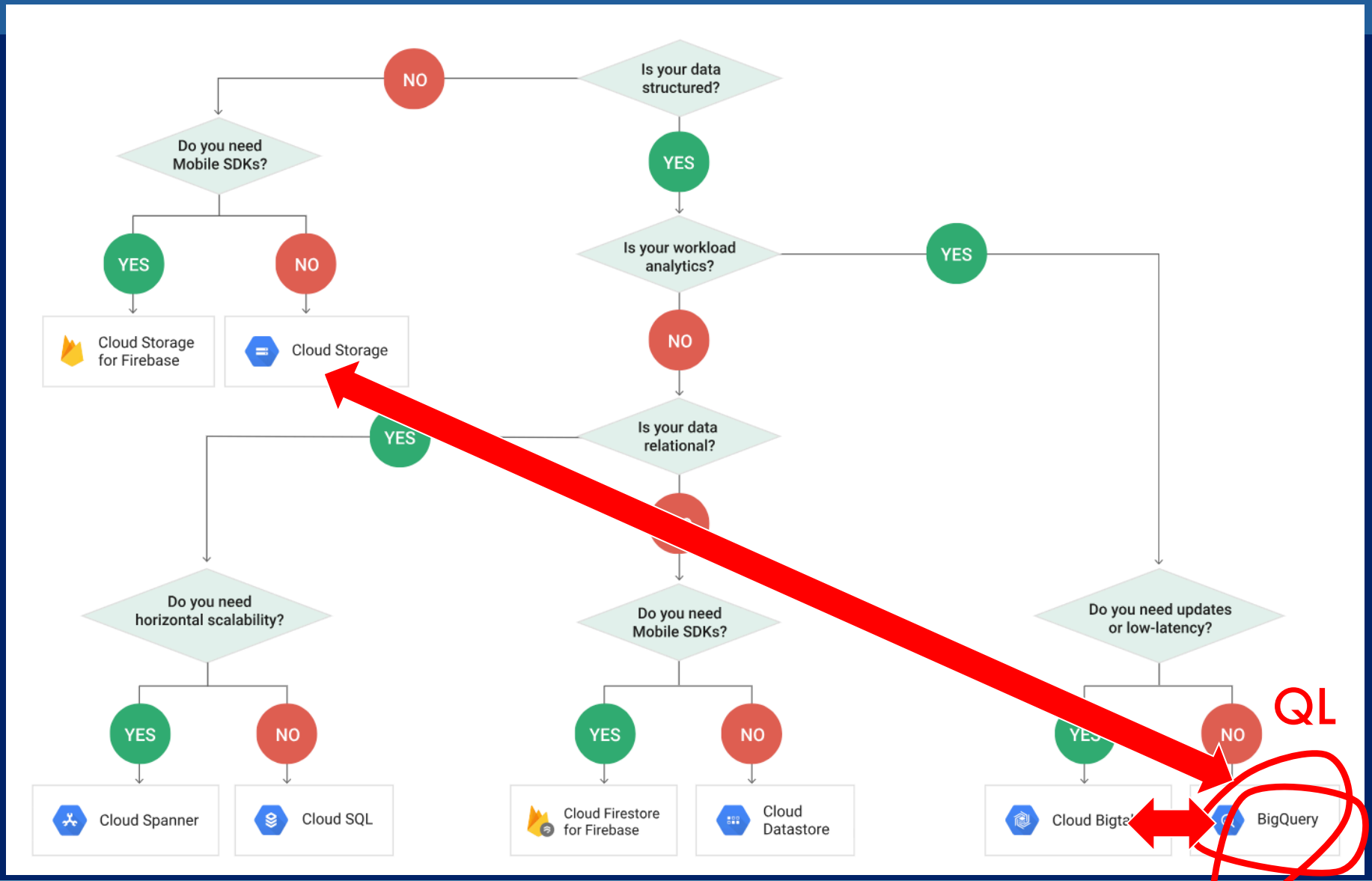
https://cloud.google.com/storage-options/



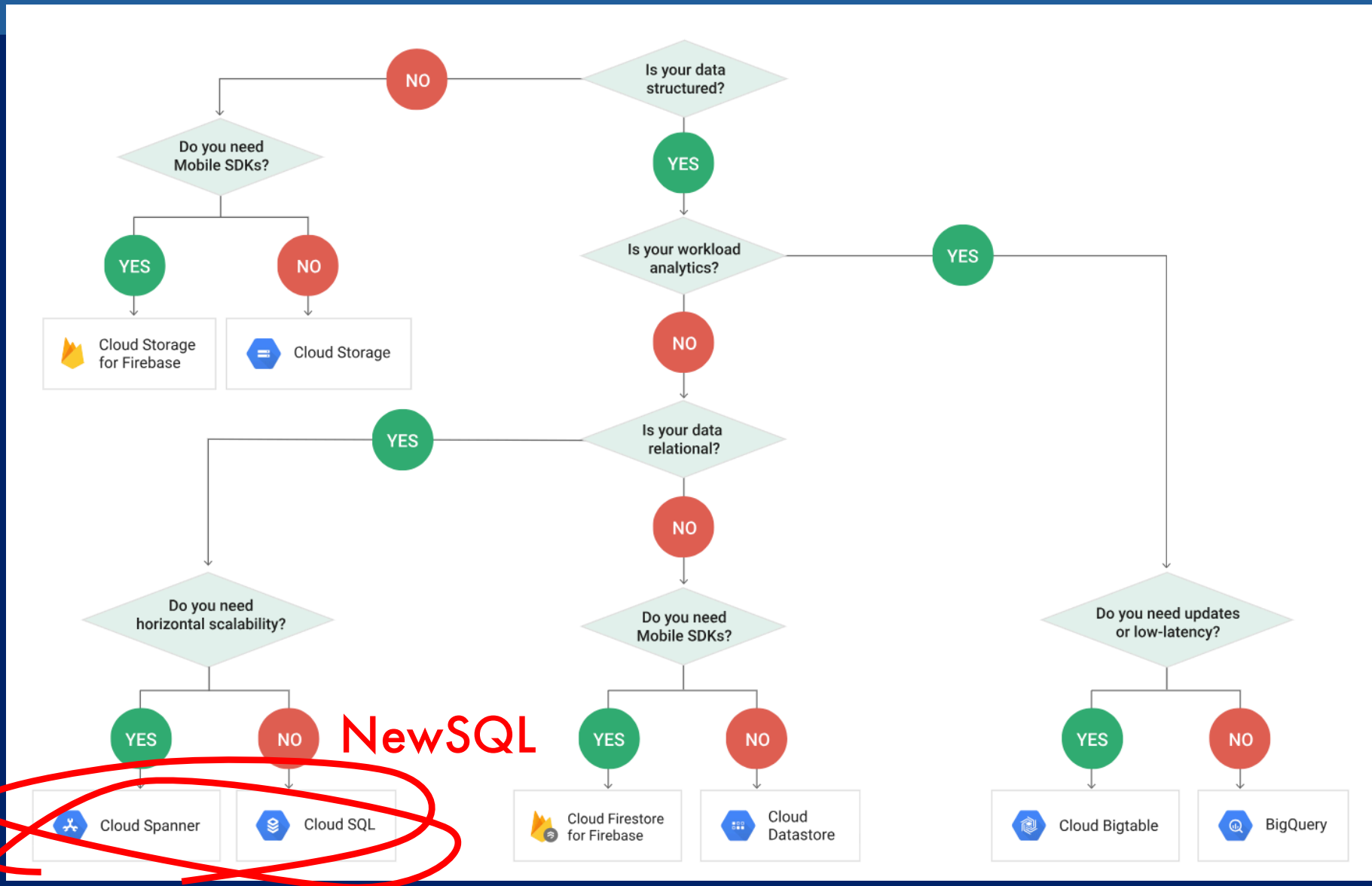
https://cloud.google.com/storage-options/



https://cloud.google.com/storage-options/



https://cloud.google.com/storage-options/



https://aws.amazon.com/products/databases/

aws

Contact Sales Support English My Account **Sign In to the Console**

re:Invent 2018 Products Solutions Pricing Learn Partner Network AWS Marketplace Explore More

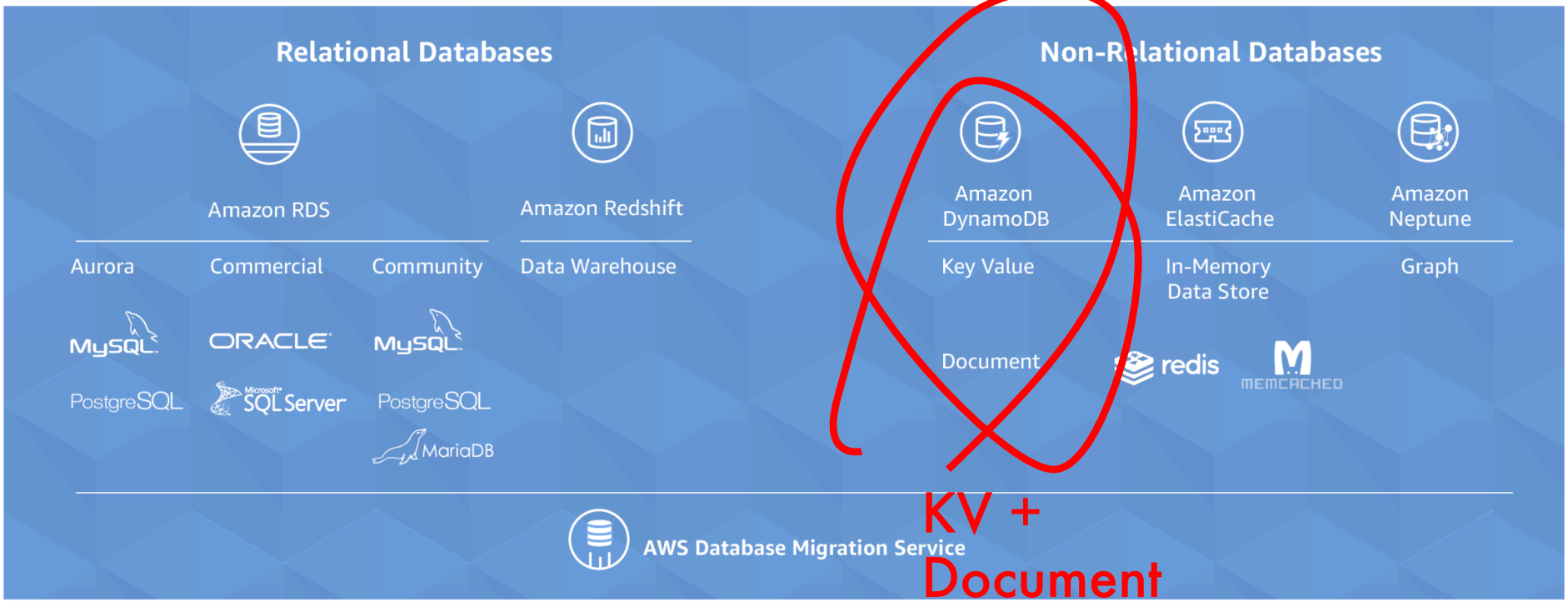
Databases on AWS

Purpose-built databases for all your application needs

As the cloud continues to drive down the cost of storage and compute, a new generation of applications have emerged, creating a new set of requirements for databases. These applications need databases to store terabytes to petabytes of new types of data, provide access to the data with millisecond latency, process millions of requests per second, and scale to support millions of users anywhere in the world. To support these requirements, you need both relational and non-relational databases that are purpose-built to handle the specific needs of your applications. AWS offers the broadest range of databases purpose-built for your specific application use cases.

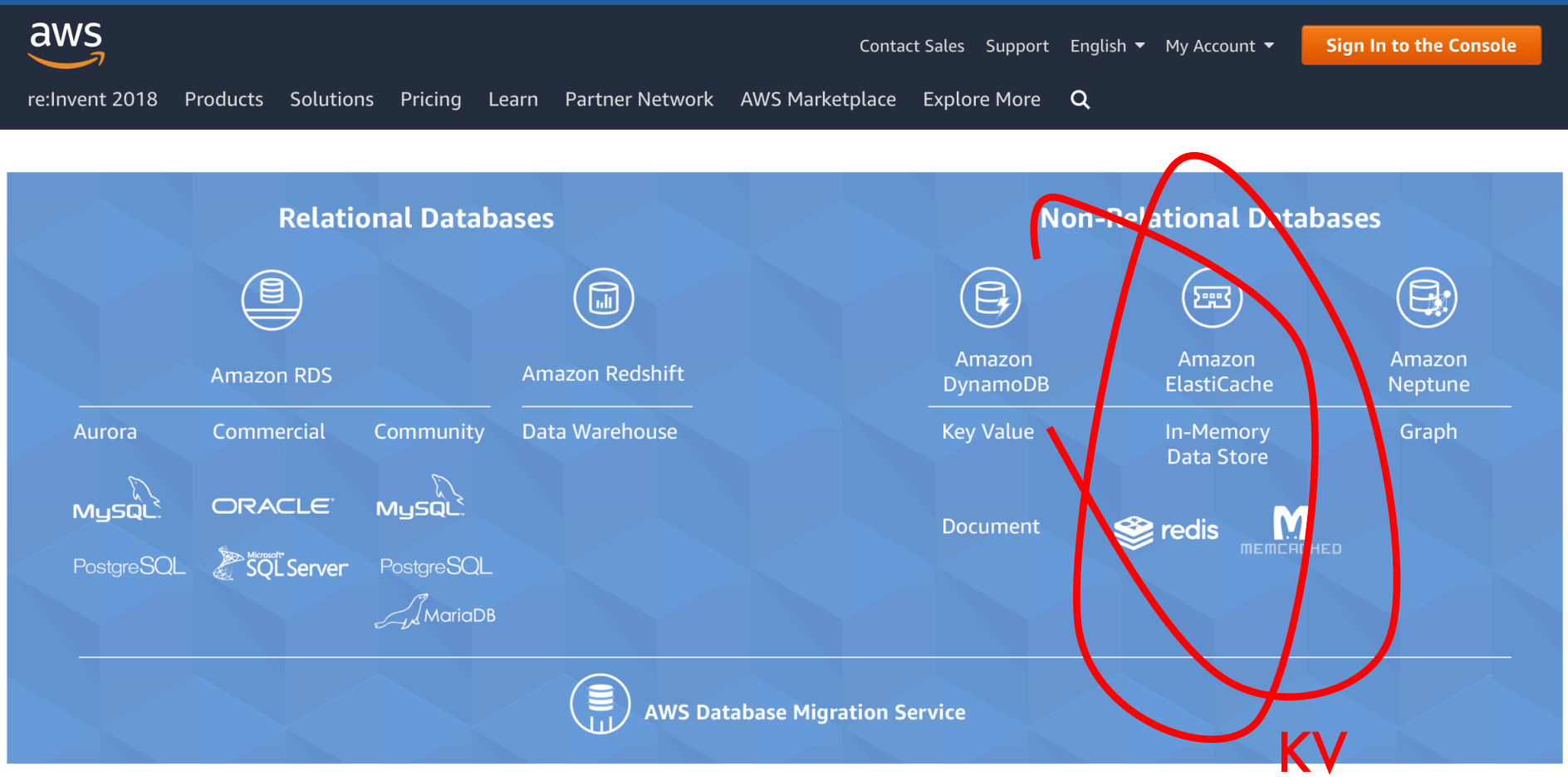
AWS Purpose-built Databases

https://aws.amazon.com/products/databases/



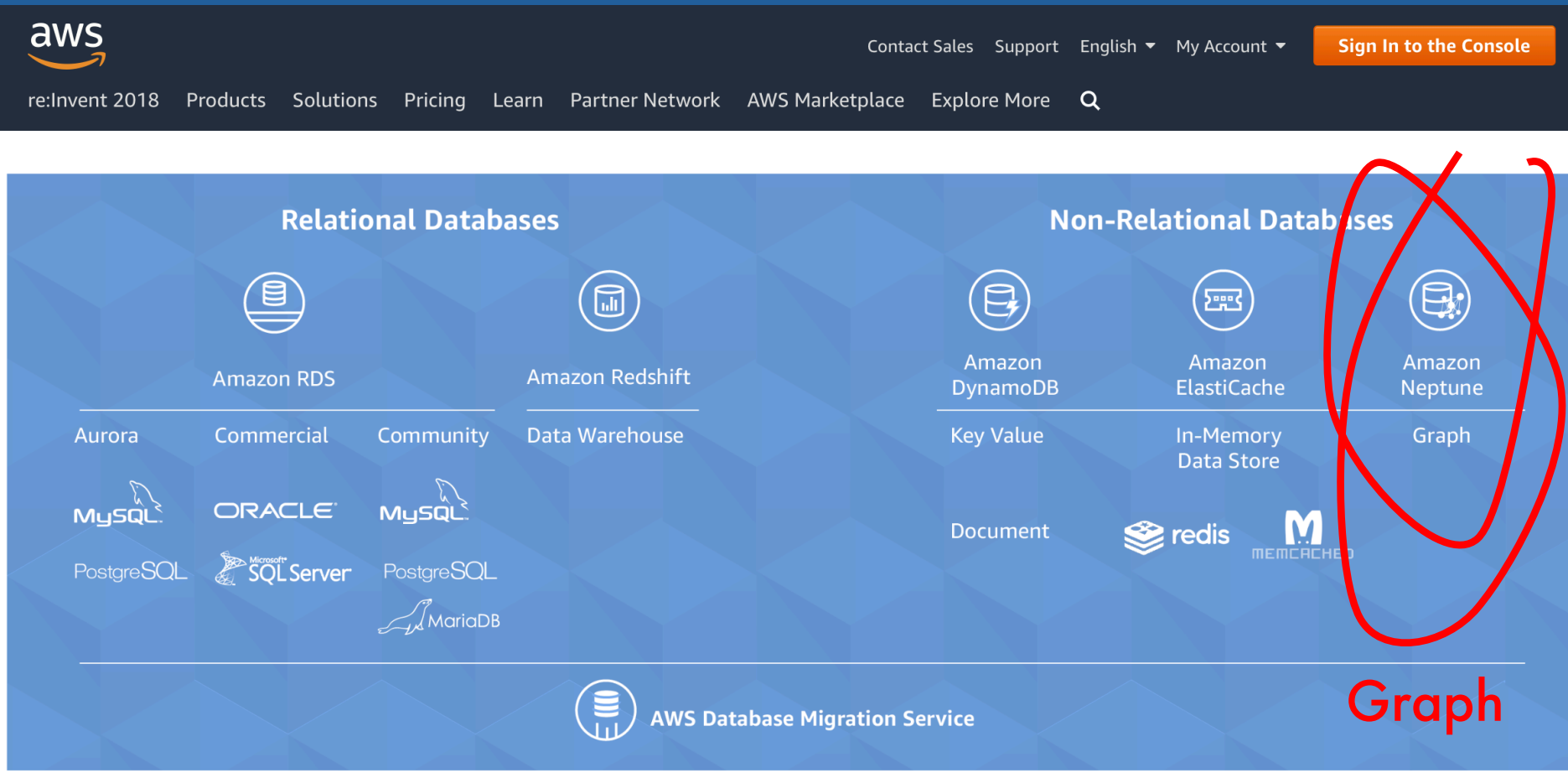
Our fully managed database services include relational databases for transactional applications, non-relational databases for internet-scale applications, a data warehouse for analytics, an in-memory data store for caching and real-time workloads, and a graph database for building applications with highly connected data. If you are looking to migrate your existing databases to AWS, the AWS Database Migration Service makes it easy and cost effective to do so.

https://aws.amazon.com/products/databases/



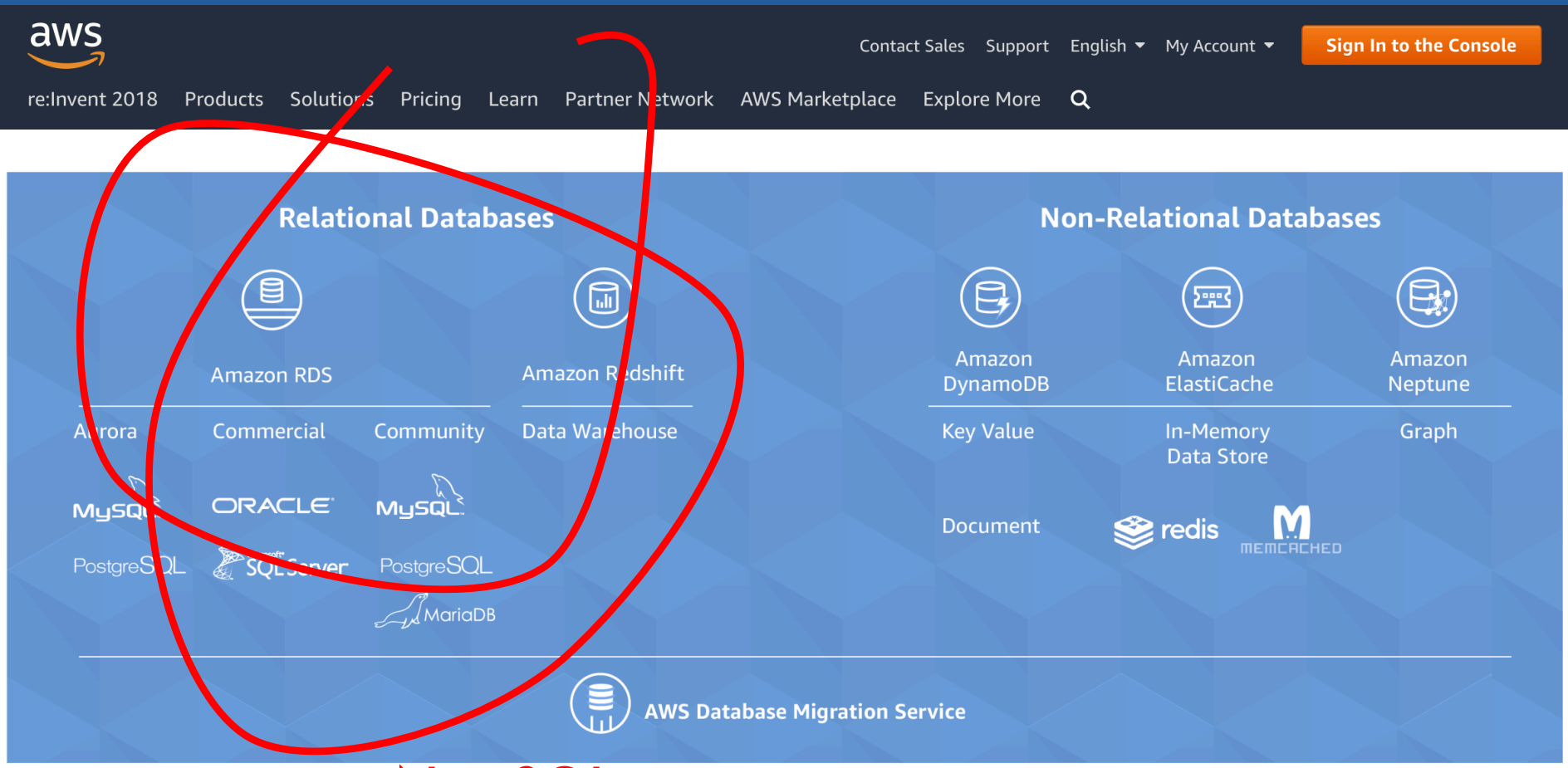
Our fully managed database services include relational databases for transactional applications, non-relational databases for internet-scale applications, a data warehouse for analytics, an in-memory data store for caching and real-time workloads, and a graph database for building applications with highly connected data. If you are looking to migrate your existing databases to AWS, the AWS Database Migration Service makes it easy and cost effective to do so.

https://aws.amazon.com/products/databases/



Our fully managed database services include relational databases for transactional applications, non-relational databases for internet-scale applications, a data warehouse for analytics, an in-memory data store for caching and real-time workloads, and a graph database for building applications with highly connected data. If you are looking to migrate your existing databases to AWS, the AWS Database Migration Service makes it easy and cost effective to do so.

https://aws.amazon.com/products/databases/



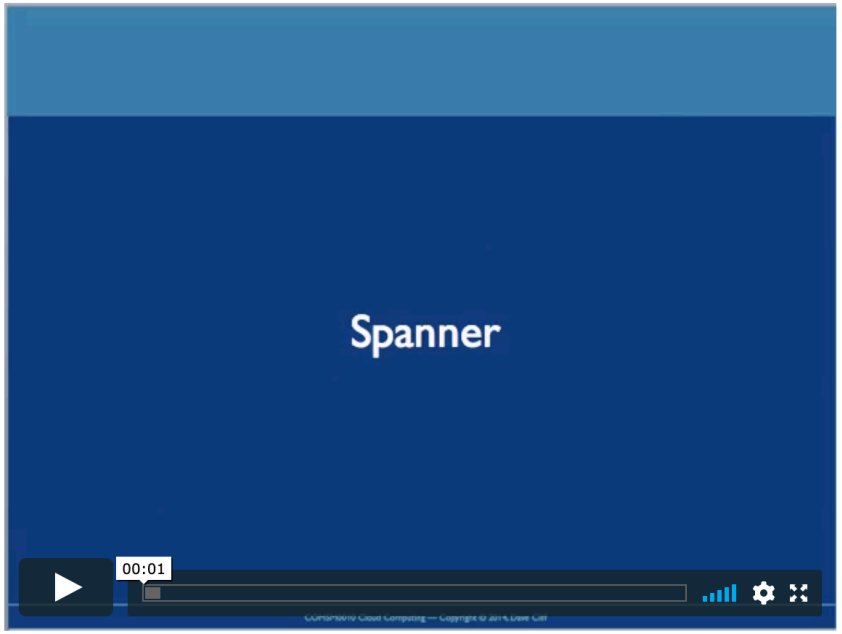
NewSQL

Our fully managed database services include relational databases for transactional applications, non-relational databases for internet-scale applications, a data warehouse for analytics, an in-memory data store for caching and real-time workloads, and a graph database for building applications with highly connected data. If you are looking to migrate your existing databases to AWS, the AWS Database Migration Service makes it easy and cost effective to do so.

NewSQL

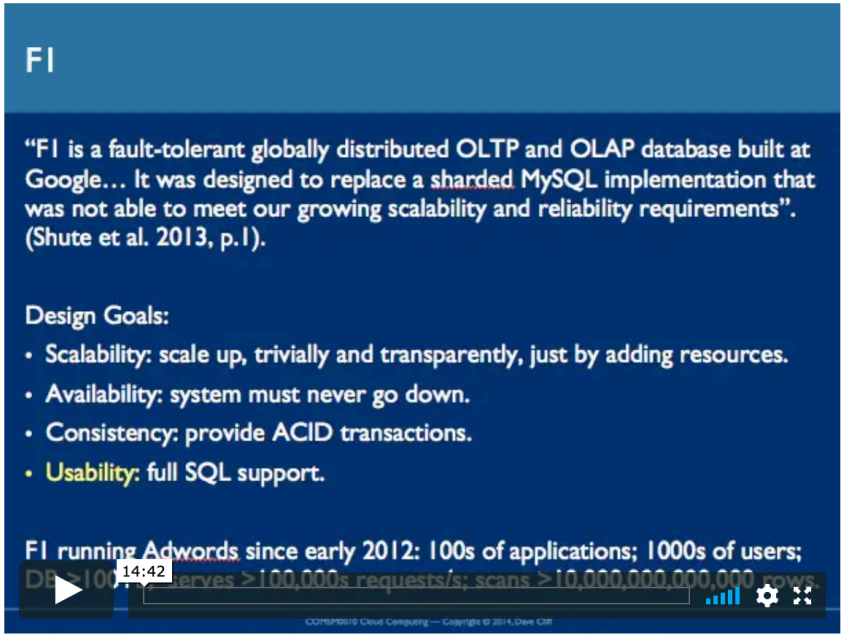
- Already covered Google *Spanner* & *F1* in earlier lecture
- Dave's videos on these are on Vimeo:

https://vimeo.com/117211077



94 0 1 0 Download

https://vimeo.com/117441822



4 0 1 0 Download

The End