NoSQL, Azure CosmosDB

Lecture 10

Deep Azure @ McKesson

Zoran B. Djordjević

History of the World, Part 1

- Relational Databases – mainstay of business
  - For a long time (and still true today), big relational database vendors such as Oracle, IBM, Sybase, and Microsoft provided Relational Database Management Systems (RDBMS) that have been at the core of our business applications
  - During the Internet boom, startups looking for low-cost RDBMS alternatives turned to MySQL and PostgreSQL.
- Web-based applications are characterized by spikes in usage
  - Could have hundreds of thousands of visitors in a short-time span. Especially true for public-facing e-Commerce sites
- Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application (ie. Ehcache)
- As datasets grew, the simple memcache/MySQL model (for lower-cost startups) started to become problematic.

History of the World, Codd

- In 1970, IBM's Edgar F. Codd released "A Relational Model of Data for Large Shared Data Banks," a paper which proposed a database structure built on the relational data model.
- A few years later, two of Codd's IBM colleagues, Donald D. Chamberlin and Raymond F. Boyce, developed "SEQUEL" (Structured English
- Query Language) – later dubbed "SQL" due to an existing trademark.
- Together, these concepts would form the foundation for the relational database.

Scaling Up

- Best way to provide ACID and a rich query model is to have the dataset on a single machine.
- There are issues with scaling up when the dataset is just too big
- RDBMS were not designed to be distributed
- Began to look at multi-node database solutions
- Known as ‘scaling out’ or ‘horizontal scaling’
- Most important approaches were
  - Master-slave
  - Sharding

Scaling RDBMS – Master/Slave

- Master-Slave
  - All writes are written to the master. All reads performed against the replicated slave databases
  - Critical reads may be incorrect as writes may not have been propagated out to replicas
  - Large data sets can pose problems as master needs time to replicate data to slaves

Scaling RDBMS - Sharding

- An alternative strategy is Partitioning or Sharding

Different sharding approaches:

- Vertical Partitioning: Have tables related to a specific feature sit on their own server. May have to rebalance or reshard if tables outgrow server.
- Range-Based Partitioning: When single table cannot sit on a single server, split table onto multiple servers. Split table based on some critical value range.
- Key or Hash-Based partitioning: Use a key value in a hash and use the resulting value as entry into multiple servers.
- Directory-Based Partitioning: Have a lookup service that has knowledge of the partitioning scheme . This allows for the adding of servers or changing the partition scheme without changing the application.
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards

Other ways to scale RDBMS

- **Multi-Master replication.**
  - The multi-master replication system is responsible for propagating data modifications made by each member to the rest of the group, and resolving any conflicts that might arise between concurrent changes made by different members.
- **INSERT only, not UPDATES/DELETES.**
  - For INSERT-only, data is versioned upon update.
  - Data is never DELETED, only inactivated.
- **No JOINs,** thereby reducing query time
  - This involves de-normalizing data
  - Consistency is the responsibility of the application.
- **In-Memory Databases**
  - In-memory databases have not caught on mainstream and regular RDBMS are more disk-intensive that memory-intensive

How did we get here?

- Explosion of social media sites (Facebook, Twitter) with large data needs. These datasets have high read/write rates.
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution) made NoSQL universally accessible
- Like a move to dynamically-typed languages (Ruby/Groovy), a shift to dynamically-typed data with frequent schema changes
- Most of the NoSQL systems, with the exception of Amazon S3 (Amazon Dynamo) and Google's BigTable are open-source solutions.

What is NoSQL?

- Should stand for **N**ot **O**nly **SQL**
- Class of non-relational data storage systems
- NoSQL systems usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties.
- For data storage, an RDBMS cannot be the only solution.
- Just as there are different programming languages, there is a need to have other data storage tools in the toolbox
- Relational databases offer a very good general purpose solution to many different data storage needs.
- Relational databases are the safe choice and will work in many situations.

Object Oriented Fizzle

- The success of NoSQL stands in contrast with an earlier challenger to RDMS dominance, the object database.
- Software projects of the pre-web era tended to integrate data rather than services, which prevented object databases from ever fully taking off.
- Attempts of some RDBMS vendors to introduce Object-Oriented model in the database itself also did not result in wide-spread adoption.
- In the post-Dot-Com Boom world, with the introduction of web APIs, the right NoSQL databases can be paired with corresponding microservices – often in multi-database architectures that leverage the most logical DB for each part of the application.

BigTable & DynamoDB

- Three major papers were the seeds of the NoSQL movement

  BigTable:

  – https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

  DynamoDB (Amazon, 2007):

  – http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

  and

  – http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

- These papers laid the groundwork for databases using new query languages and more flexible schemas that would provide greater scalability, speed and developer ease of use.
- In 2009, a meet-up in San Francisco gave a single identity to these various databases springing up. NoSQL started as a hashtag for that meet-up, but it later grew to encompass all databases that don't use the SQL query language

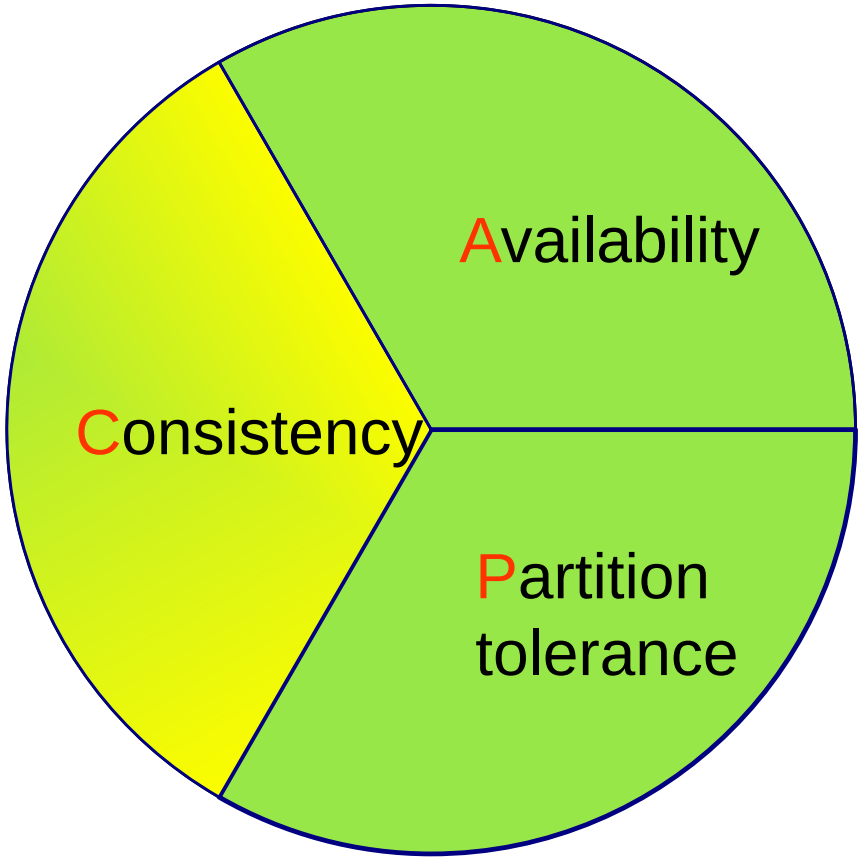
The Perfect Storm

- Large datasets, acceptance of alternatives, and dynamically-typed data has come together in a perfect storm.
- Industry have reached a point where a read-only cache and write-based RDBMS isn't delivering the throughput necessary to support many internet scale application.
- Not a backlash/rebellion against RDBMS
- The NoSQL databases are a pragmatic response to growing scale of databases and the falling prices of commodity hardware.
- Most likely, 10 years from now, majority of transactions related data will still be stored in RDBMS. Non-transactional data are (mostly) flowing to NoSQL storage.
- SQL is a rich query language that could not be pushed out by the current list of NoSQL offerings

CAP Theorem

- Proposed by Eric Brewer (talk on Principles of Distributed Computing July 2000).
- Three properties of a system: **C**onsistency, **A**vailability and **P**artitionability.
- **You can have at most two of these three properties for any shared-data system**
  - **Partitionability**: Can divide nodes into small groups that can see other groups, but they can't see everyone.
  - **Consistency**: write a value and then you read the value you get the same value back. In a partitioned system there are windows where that's not true.
  - **Availability**: may not always be able to write or read. The system will say you can't write because it wants to keep the system consistent.
- To scale you have to partition, so you are left with choosing either high consistency or high availability for a system. Find the right overlap of availability and consistency. Choose an approach based on your service
- For the checkout process, you always want to honor requests to add items to a shopping cart because it's revenue producing. In this case you choose high availability. Errors are hidden from the customer and sorted out later.
- When a customer submits an order you favor consistency because several services--credit card processing, shipping and handling, reporting— are simultaneously accessing the data.

Consistency, The CAP Theorem View



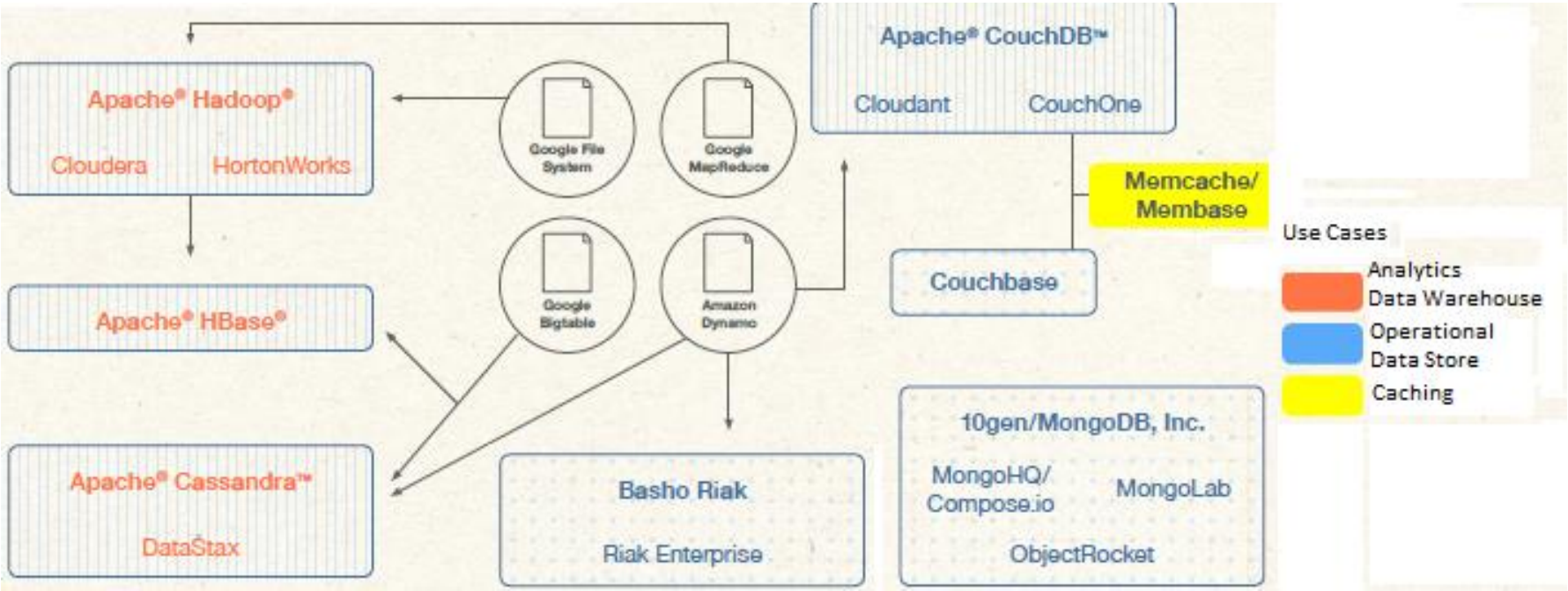
Once a writer has written, all readers will see that write.

- Two kinds of consistency:
  - strong consistency – ACID(Atomicity Consistency Isolation Durability)
  - weak consistency – BASE(Basically Available Soft-state Eventual consistency)

NoSQL Family Tree

- Modern NoSQL Databases include:
  - key-value,
  - column family,
  - document and
  - graph stores
- as the major types.

- These systems are popular because they more naturally represent data structures used in software development.
- Hadoop (MapReduce) Eco system made several notable contributions to NoSQL databases, i.e. warehouses: Cloudera Impala, Apache Hive, others.



Examples of NoSQL Databases

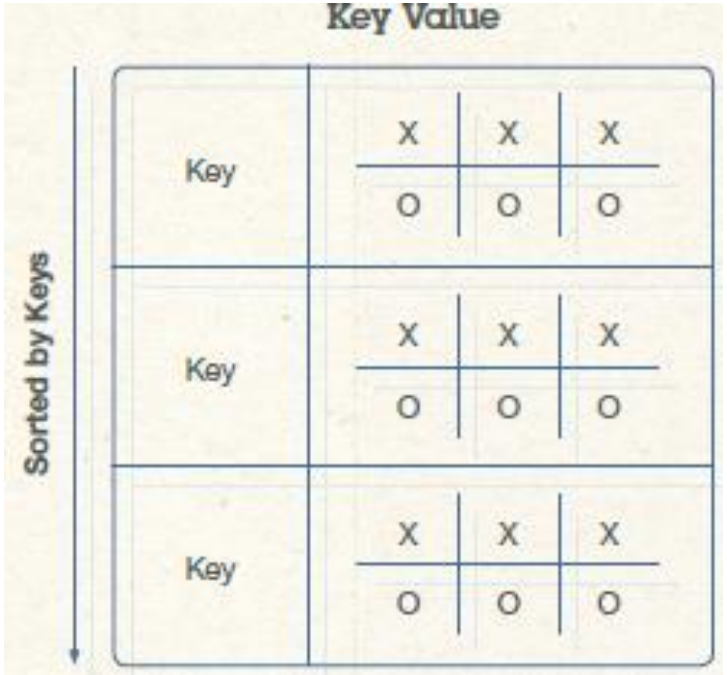
- Key/Value or 'the big hash table'.
  - Amazon S3 (Dynamo)
  - Voldemort
  - Scalaris
  - Memcached (in-memory key/value store)
  - Redis
- Schema-less which comes in multiple flavors, column-based, document-based or graph-based.
  - Cassandra (column-based)
  - CouchDB (document-based, document: views are stored as rows which are kept sorted by key.)
  - MongoDB(document-based)
  - Neo4J (graph-based, is a network database that uses edges and nodes to represent and store data)
  - HBase (column-based)

Key-Value Stores

- IBM's Indexed Sequential Access Method (ISAM) of the 1970s, is actually the oldest key-value model.
- Key-value represents the most basic type of non-relational database where each item in the database is stored as an attribute name - or key - together with its value.
- Another popular key-value database, Berkeley DB, originated at the UC-Berkeley Computer Systems Research Group in the early 1990s, and is still in wide use.
- It was Amazon's Dynamo 2007 whitepaper that sparked the use of key-value stores in distributed systems. One significant takeaway from the whitepaper was that Dynamo enables high availability. When several copies of a database are spread across many nodes, the process by which they coordinate their response is known as "quorum."
- With a Dynamo-style quorum (used in AWS DynamoDB, IBM Cloudant or Riak), the process can be tuned to provide looser or stricter levels of consistency.
- Typically, a relaxed consistency model means that if part of a cluster goes down, the data can still be accessed and consistency can be coordinated when all nodes are back online. For many applications, it's often better to be able to access stale data than having no access at all.

Schema-less Storage

- Key-value stores also allow the application developer to store schema-less data.
- Key-value databases were born from the need to support rapid scaling data, and are commonly used in scenarios requiring a constant stream of small reads and writes, such as user preference or profile stores, product recommendations and real-time digital advertising.
- Key-value stores are extremely useful for applications that only query data by a single key.
- Performance and scalability, combined with the simplicity of data access patterns, are the main attractions of these types of databases.



Get, Put, Delete

- In the key-value structure, the key is usually a simple string of characters, and the value is a series of uninterrupted bytes that are opaque to the database. The data itself is usually some primitive data type (string, integer, array) or a more complex object that an application needs to persist and access directly.
- This replaces the rigidity of relational schemas with a more flexible data model that allows us to easily modify fields and object structures as the apps evolve.
- Some key-value databases, such as Redis, support familiar data structures, including lists, sets, hashes and sorted sets. They enable users to read and write values by using a key:
  - Get (key), returns the value associated with the provided key
  - Put (key, value), associates a value with the key
  - Multi-get (key1, key2, etc.), returns the list of values associated with the list of keys
  - Delete (key), removes the entry
- Initially, key-value stores had no query language. They simply provide a way to store, retrieve and update data using simple get, put and delete commands. The path to retrieve data is a direct request to the object, whether it is in memory or on disk. The simplicity of this model makes a key-value store fast, easy to use, scalable, portable and flexible.

Simplicity, Usage

- A key-value store consists of a set of tables with keys paired to objects (values), like a hash table in computer programming.
- There's no comparison, aggregation or sorting of records. Since values are opaque, we do not index the data to improve performance. You cannot, however, filter or control what's returned from a request based on the value.
- Key-value stores scale out by implementing sharding, replication and auto recovery. They scale up by maintaining the database in RAM and minimizing the effects of ACID guarantees (a guarantee that committed transactions persist somewhere) by avoiding locks, latches and low-overhead server calls.
- Key-value stores are generally good solutions if you have a simple application with only one kind of object, and you only need to look up objects based on one attribute. The simplicity of key-value stores also may make them the easiest to use.
- Say you're running a search engine, and lots of people are searching for news around a particular topic. Rather than the same queries hitting the database over and over again, the queries and their corresponding result sets can be cached in a key-value store, where the query string is the "key" and the "value" is a list of relevant news articles. That way, the database is better able to respond to new queries and stash results in the cache. Thus, the key-value store acts as a cache, reducing read-load on the RDBMS server.

Document Stores

- A document store does not have much to do with “documents” in the sense of a MS Word document or a PDF document.
- The document refers to a data record that is self-describing in regards to the data elements it contains. XML is a self-describing and JSON is self-describing.
- A document is an object and keys (strings) can have values of recognizable types, including numbers, Booleans, strings, as well as nested arrays and dictionaries.
- Documents can also contain BLOBs, called attachments. You can add, change or remove attachments in a very similar way as you would add, change or remove key-value data in a doc.
- At its core, a document database can be considered a key-value store with one major exception: Instead of persisting opaque values, a document database requires the data to be stored in a format that the database can understand (i.e. JSON, XML, etc.).
- Some document databases require a driver to convert the binary representation of the on-disk data into these formats. Others store data natively in formats like JSON.
- Initially with these databases, in which no driver was required, the only way to talk to the database was by using its HTTP API. Once the data is in the right format, you can enable queries on the data’s values.

Simplicity and Scalability

- The overall concerns document databases address are simplicity and scalability, as well as fast iteration in development.
- Say you need to add a new field to your object to run a new feature in your app. With a doc store, there's no schema to update – just add the new field and go. There is no messing with ORMs or worrying about breaking someone else's feature.

Document View, all info in a single document

In Relational World, you use at least 2 tables

No Schema, No Issue with Change

- Document databases offer an alternative to relational databases, not a replacement.
- When developing an application with a relational database, all the app's data elements are first mapped to an abstract entity-relationship model, which defines the data, relationships and structure that will then subsequently be used to create relational table and column definitions.
- If (and when) the application data model changes, each of the corresponding table and column definitions need to be adjusted accordingly. Typically, these types of changes require the application developer to request the database administrator to aid in updating the schema.
- For example a User Profile is stored in an RDBMS is broken apart into its components, each then stored in a unique table for that component. (Here, the component is the data's unit of aggregation.)
- If you would like to extract complete information about a User, the parts would need to be retrieved from the tables via JOINs and then put together into the combined result the user wants.
- If you want to make changes to the structure of user profile, we would need to modify one or many tables, unload and reload the data.
- A document database, in which information is stored and accessed in a more aggregated form, is probably a better option. In a document store, an entire user profile would be stored as the unit of aggregation, and can be retrieved as a single unit (document), without the need to do all the JOINs like you'd have to do with an RDBMS.

Graph Databases

- A graph database is another example of technology created to fix a problem that relational databases simply were not designed to handle.
- The modern graph database, pioneered by Neo4j inventor Emil Eifrem, is a data storage and processing engine that makes the persistence and exploration of data and relationships more efficient.
- In graph theory, structures are composed of vertices and edges (data and connections), or what would later be called “data relationships.”
- Graphs behave similar to how people think – in specific relationships between discrete units of data.
- Traditional database management systems store data in tabular form – rows and columns – but the information we deal with on a daily basis – including our knowledge of family members, coworkers and bank accounts – exists in relationships.
- Why shoehorn data into tabs and rows if it only slows the work and complicates or restricts the analysis?
- By storing data relationships directly as a graph – made up of nodes or vertices, connected via relationships or edges to form a mesh of information – we reduce complexity and eliminate the extra work involved in transforming the data from the model to storage.

Relationships

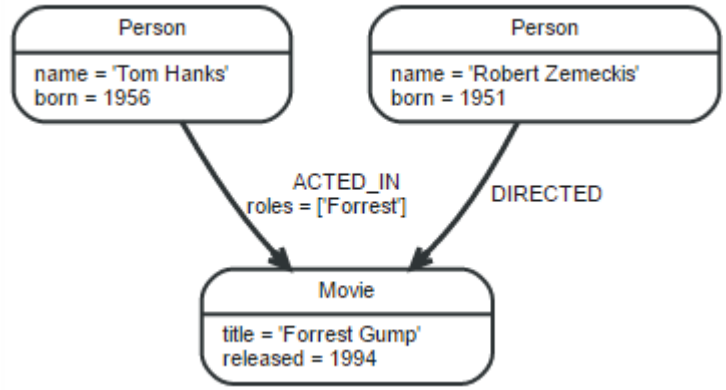
- Querying graphs is all about traversal. Graph queries answer questions like, “Can I find a path through the graph that starts here and ends there?” and “What can I learn from the connections along that path?”
- Think of a map, in which the nodes are street intersections and the edges are the roads connecting them. Each section of road probably has a name, a type, a length, a speed limit, etc. What do you learn from piecing together that path?
- When you query a graph database, you get all the nodes (data points) that have a particular property and are related to other nodes.
- Both nodes and edges can store additional properties such as key-value pairs. This is important because all of the data in your organization is only important in how it relates to other data points.
- The value of the data rests in data relationships; graph databases let you get at that value more quickly and easily. Graph visualization of data relationships leads to a fuller picture of what’s happening, so that users have better insight.
- Another key use for graph databases is to serve up “rich” in-app recommendations, drawing from both current session and historical data, along with any personalization and sentiment analysis that may already exist about the user. This combination used to be tricky to pull off because it involved utilizing multiple different data stores and
- usually different data types.

Key Concepts

- Key concepts in a graph database are:
  - Nodes
  - Relationships
  - Properties
  - Labels
  - Traversal
  - Paths
  - Schema

- A graph records data in nodes and relationships. Both can have properties.
- This is sometimes referred to as the *Property Graph Model*.

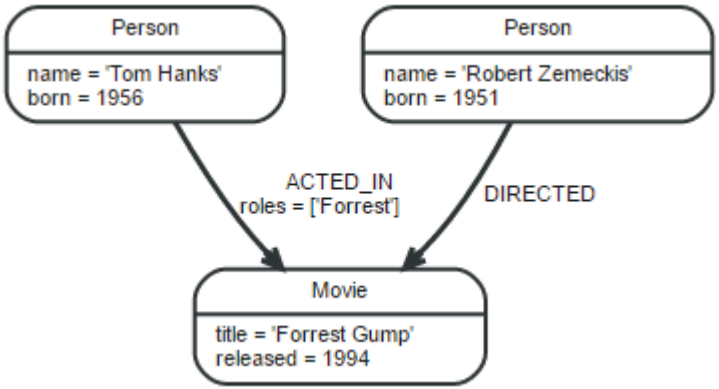
- A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way. The following is an example of a simple graph:



- This graph contains three nodes, two with label Person and one with Label Movie.
- One person is related to the move since he DIRECTED it and the other ACTED_IN it.
- Nodes and relationships have properties.

Relationships

- Relationships organize the nodes by connecting them. A relationship connects two nodes — a start node and an end node. Just like nodes, relationships can have properties.
- Relationships between nodes are a key part of a graph database. They allow for finding related data. Just like nodes, relationships can have properties.
- A relationship connects two nodes, and is guaranteed to have valid start and end nodes.
- Relationships organize nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which can be combined into yet more complex, richly inter-connected structures.



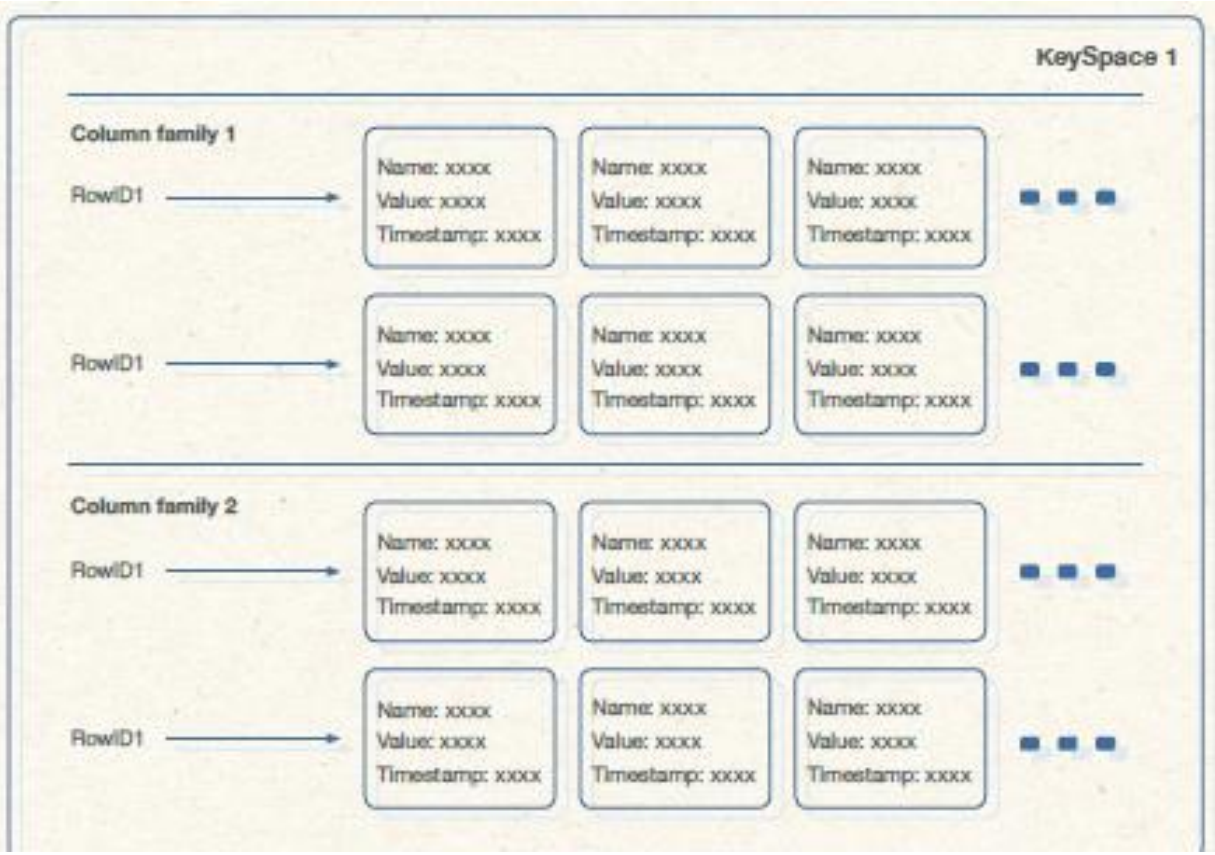
- Graph to the left uses ACTED_IN and DIRECTED as relationship types.
- The roles property on the ACTED_IN relationship has an array value with a single item in it.
- ACTED_IN relationship has Tom Hanks node as start node and Forrest Gump as end node.
- Tom Hanks node has an outgoing relationship, while the Forrest Gump node has an incoming relationship

- Relationships are equally well traversed in either direction.
- There is little need to add duplicate relationship in the opposite direction.

Column (Family) Stores

- Column family stores – like Apache Cassandra and Apache HBase – share a similar architectural structure with key-value stores, but are otherwise quite different from other non-relational databases.
- "Column family" refers to the sets of columns that are the units of access control in this type of data store.

Cassandra
Data Model



Characteristics of Column Family databases

- Column family stores utilize hash maps (essentially a list of key-value pairs). They are organized into cells in corresponding columns. A record is a grouping of these columns. It's possible for columns to be "sparse," which means there is no schema forcing each record to have a corresponding entry in every column.
- Column family stores enable very quick data access using a row key, column name and cell timestamp.
- **Characteristics of column family databases include:**
  - High scalability and high availability
  - No single point of failure
  - High write throughput and good read throughput
  - Flexible schemas
  - Tunable consistency
  - Support for replication
  - **No support for ACID transactions** – there is no guarantee that batch operations will be carried out in an atomic fashion.
  - **No support for JOINS** – if you need to join two column families, you must do it programmatically. However, because they store so much data in single columns, column family stores often circumvent the need for JOINs.
  - **Keys must be unique** – if the same key has been used twice, it will overwrite data.
  - **Secondary search indexes may not be supported** – you must instead build indexes using sort orders and slices.

Columnar or Column-oriented Database

- Column family stores and columnar databases (also called “column-oriented” databases) are not the same. They might share similar names and a few structural elements, but they’re two independent concepts.
- As veteran DBAs will notice, columnar databases are actually a flavor of relational databases, but with data organized in columns, not rows, for querying.
- Column family data tables are stored in columns.
- In columnar databases, the columns aren’t homogenous across rows.

- Examples of columnar databases are: Sybase IQ, and Vertica.

Columnar (Column-oriented) Systems

- Important column-oriented databases have been present for a while(Vertica, 2005; Statistics Canada RAPID System, 1969).
- A column-oriented database serializes all of the values of a column together, then the values of the next column, and so on. For our example table, the data would be stored in this fashion:

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

- Any one of the columns more closely matches the structure of an index in a row-based system. This creates an impression that column-oriented store "is really just" a row-store with an index on every column.
- It is the mapping of the data that differs dramatically. In a row-oriented indexed system, the primary key is the rowid that is mapped to indexed data.

Data organization in Column-oriented systems

- In the column-oriented system primary key is the data, mapping back to rowids. The difference can be seen in the case when we have two rows (users) with the same last name. Column “last_name” would be stored as:

```
…;Smith:001,Jones:002,004,Johnson:003;…
```

- Record `"Jones"` could be saved only once in the column store along with pointers to all of the rows that match it. For many common searches, like "find all the people with the last name Jones", the answer is retrieved in a single operation. Similarly, counting the number of matching records, can be greatly improved through this organization.
- In a column-oriented system operations that retrieve complete data for objects would be slower, requiring numerous disk operations to collect data from multiple columns to build up the record.
- In the many cases, only a limited subset of data is retrieved. If we are collecting the first and last names from many rows in order to build a list of contacts, columnar organization is vastly beneficial.
- This is even more true for writing data into the database, especially if the data tends to be "sparse" with many optional columns.

Benefits of Column-oriented DBs

- Column-oriented organizations are more efficient when an aggregate needs to be computed over many rows but only for a notably small subset of all columns of data
- Column-oriented organizations are more efficient when new values of a column are supplied for all rows at once, because that column data can be written efficiently and replace old column data without touching any other columns for the rows.
- Row-oriented organizations are more efficient when many columns of a single row are required at the same time, and when row-size is relatively small, as the entire row can be retrieved with a single disk seek.
- Row-oriented organizations are more efficient when writing a new row if all of the row data is supplied at the same time, as the entire row can be written with a single disk seek.
- In practice, row-oriented storage layouts are well-suited for OLTP-like workloads which are more heavily loaded with interactive transactions. Column-oriented storage layouts are well-suited for OLAP-like workloads.

Common Advantages of NoSQL Systems

- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - **Down nodes easily replaced**
  - **No single point of failure**
  - **As the data is written, the latest version is on at least one node. The data is then versioned/replicated to other nodes within the system.**
- Eventually, the same version is on all nodes.
- Easy to distribute
- Don't require a schema
- Can scale up and down
- Relax the data consistency requirement (CAP)

Row Oriented Databases

- A relational database management system maintains data that represents two-dimensional tables, with columns and rows. For example, a database might have table Employee:

EmpId	Lastname	Firstname	Salary
10	Smith	Joe	40000
12	Jones	Mary	50000
11	Johnson	Cathy	44000
22	Jones	Bob	55000

- This two-dimensional format exists only on paper. Storage hardware requires the data to be serialized into a sequence of “cells” and placed onto the hard drives.
- The most expensive operations involving hard drives are seeks. In order to improve overall performance, related data should be stored in a fashion to minimize the number of seeks. Hard drives are organized into a series of blocks of a fixed size, typically enough to store several rows of the table. This minimizes the number of data retrievals.

Row Organized Data

- The common solution to the storage problem is to serialize each row of data, and assign to it a row id. Rows of the previous table could be packaged like this:

```
001:10,Smith,Joe,40000;002:12,Jones,Mary,50000;003:11,Johnson,Cathy,44000;004:22,Jones,Bob,55000;
```

- Indicators `001, 002, 003` and `004` represent row ids. In practice, those identifiers are usually longer, 64-bit or 128-bit strings.
- In OLTP systems, we need the entire raw(s) of data in order to populate entire object(s). It makes every sense to store all components of a row of data together. By storing the record's data in a single block on the disk, along with related records, the system can quickly retrieve records with a minimum of disk operations.
- Row-based systems are not efficient at performing operations that apply to the entire data set, as opposed to a specific record. For instance, in order to find all the records in the example table that have salaries between 40,000 and 50,000, the DBMS would have to seek through the entire data set looking for matching records.

Indexes Help

- To improve the performance of these sorts of operations, most DBMS's support the use of database indexes, which store all the values from a set of columns along with pointers back into the original rowid.
- An index on the salary column would look something like this:

```
001:40000;002:50000;003:44000;004:55000;
```

- As they store only single pieces of data, rather than entire rows, indexes are generally much smaller than the main table stores. By scanning smaller sets of data the number of disk operations is reduced. If the index is heavily used, it can provide dramatic time savings for common operations. However, maintaining indexes adds overhead to the system, especially when new data is written to the database. In this case not only is the record stored in the main table, but any attached indexes have to be updated as well.
- Database indexes on one or more columns are typically sorted by value, which makes operations like range queries fast.

Modern RDBMS

- There is a number of row-oriented databases that are designed to fit entirely in RAM, the so called an in-memory database (Oracle Times Ten).
- RAM is rapidly getting cheaper and big vendors like Oracle, (Microsoft,) IBM are offering specialized hardware with enormous RAM-s (several TB-s)
- These systems do not depend on disk operations, and have equal-time access to the entire dataset. This reduces the need for indexes, as it is required the same amount of operations to full scan the original data as a complete index for typical aggregation purposes. Such systems are simpler and smaller, but can only manage databases that fit into the memory.
- Classical Hard Drives are being replaced by Solid State Drives (SSD-s, Flash Drives) and several vendors (Aerospike, Amazon DynamoDB) are rewriting RDBMS systems to take advantage of new technology. Consistent reads and writes complete in under 1 millisecond on such systems. This is two orders of magnitude faster than on the classical HD systems.
- Traditional databases also try to scale with volume of data by using cluster technology. For whatever reason there are limitations to such scaling.
- **RDBMS are far from dead. They will remain the backbone of all IT systems.**

Compression

- Column data is of uniform type. Many popular modern compression schemes, such as LZW or run-length encoding, make use of the similarity of adjacent data to compress. While the same techniques may be used on row-oriented data, a typical implementation will be less effective.
- To improve compression, sorting rows can also help. For example, using bitmap indexes, sorting can improve compression by an order of magnitude. To maximize the compression benefits of the lexicographical order with respect to run-length encoding, it is best to use low-cardinality columns as the first sort keys. For example, given a table with columns sex, age, name, it would be best to sort first on the value sex (cardinality of two), then age (cardinality of <150), then name.
- Columnar compression achieves a reduction in disk space at the expense of efficiency of retrieval. Retrieving all data from a single row is more efficient when that data is located in a single location, such as in a row-oriented architecture. Further, the greater adjacent compression achieved, the more difficult random-access may become, as data might need to be uncompressed to be read.

Benchmark Configuration, DataStax Evaluations

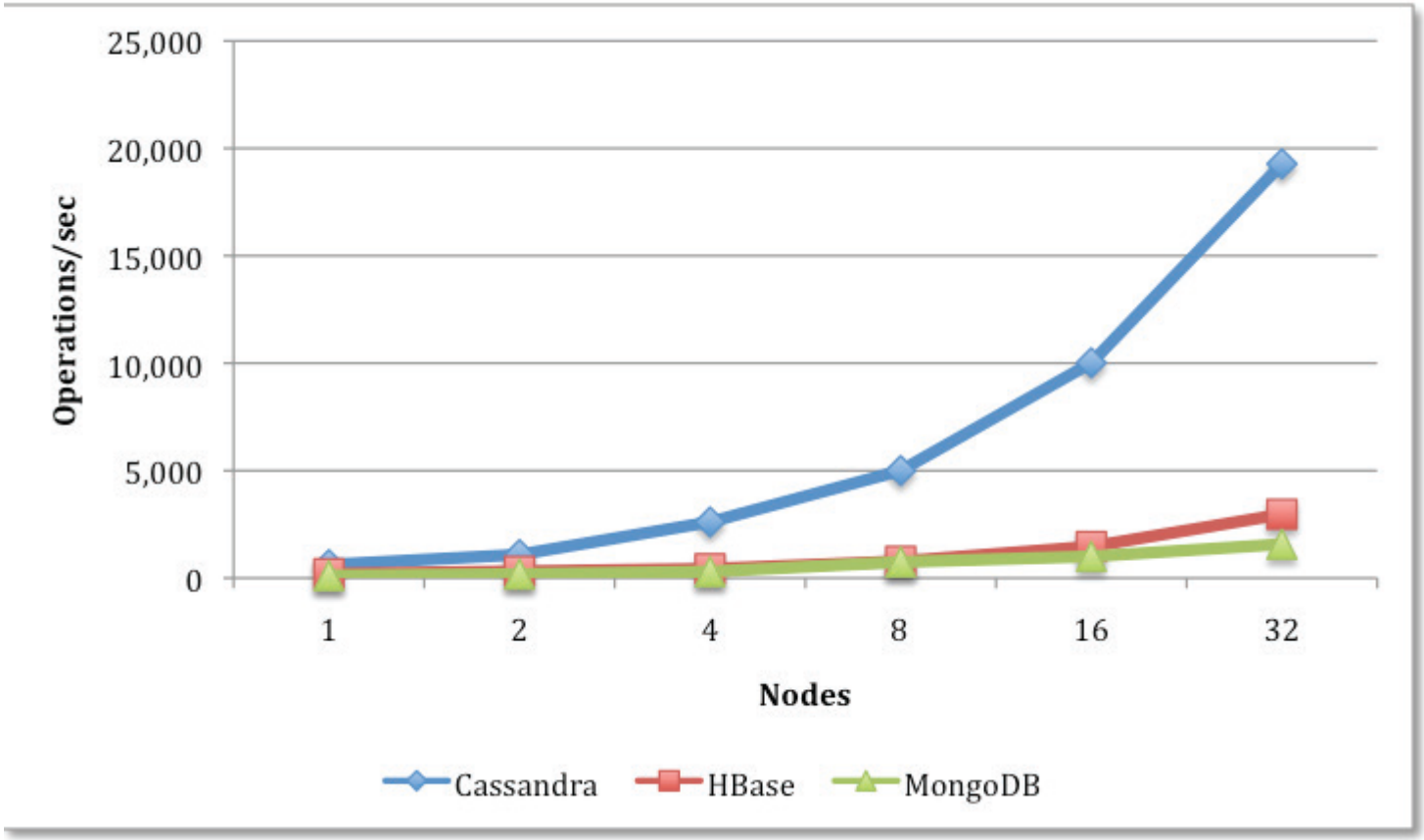
- The tests ran in the cloud on Amazon Web Services (AWS) EC2 instances, using spot instances to ensure cost efficiency while getting the same level of performance.
- The tests ran exclusively on m1.xlarge size instances (15 GB RAM and 4 CPU cores) using local instance storage for performance. The m1.xlarge instance type allows for up to 4 local instance devices; the instances were allocated all 4 block devices, which were then combined on boot into a single 1.7TB RAID-1 volume.
- The instances use customized Ubuntu 12.04 LTS AMI's with Oracle Java 1.6 installed as a base.
- On start up, each instance calls back to a parent instance for its configuration. A customized script was written to drive the benchmark process, including managing the start up, configuration, and termination of EC2 instances, calculation of workload parameters, and driving the clients to run the tests.

Tested Workloads

The following workloads were included in the benchmark:

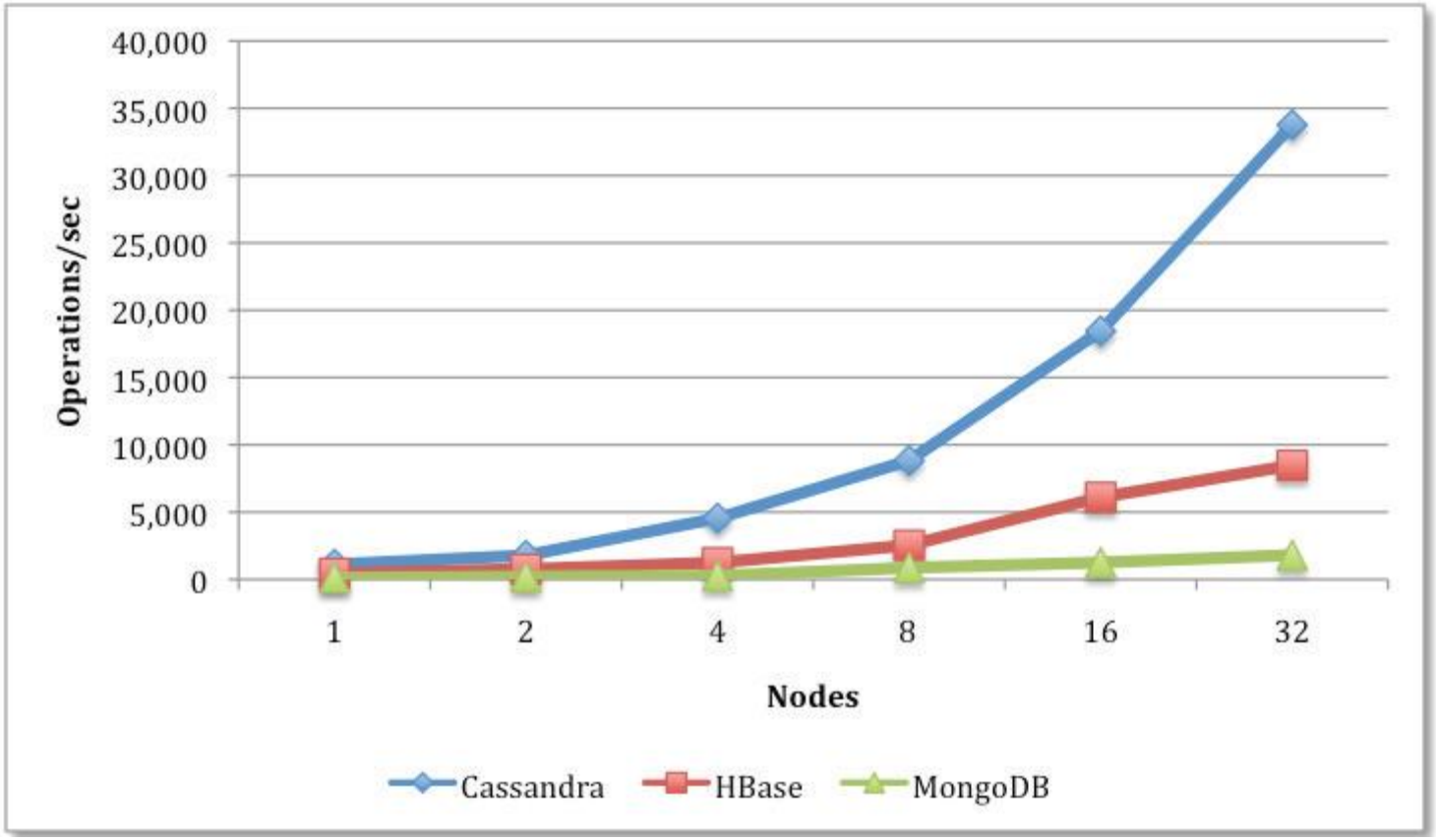
1. Read-mostly workload, based on YCSB's provided workload B: 95% read to 5% update ratio
2. Read/write combination, based on YCSB's workload A: 50% read to 50% update ratio
3. Write-mostly workload: 99% update to 1% read
4. Read/scan combination: 47% read, 47% scan, 6% update
5. Read/write combination with scans: 25% read, 25% scan, 25% update, 25% insert
6. Read latest workload, based on YCSB workload D: 95% read to 5% insert
7. Read-modify-write, based on YCSB workload F: 50% read to 50% read-modify-write

Read-Mostly Workload



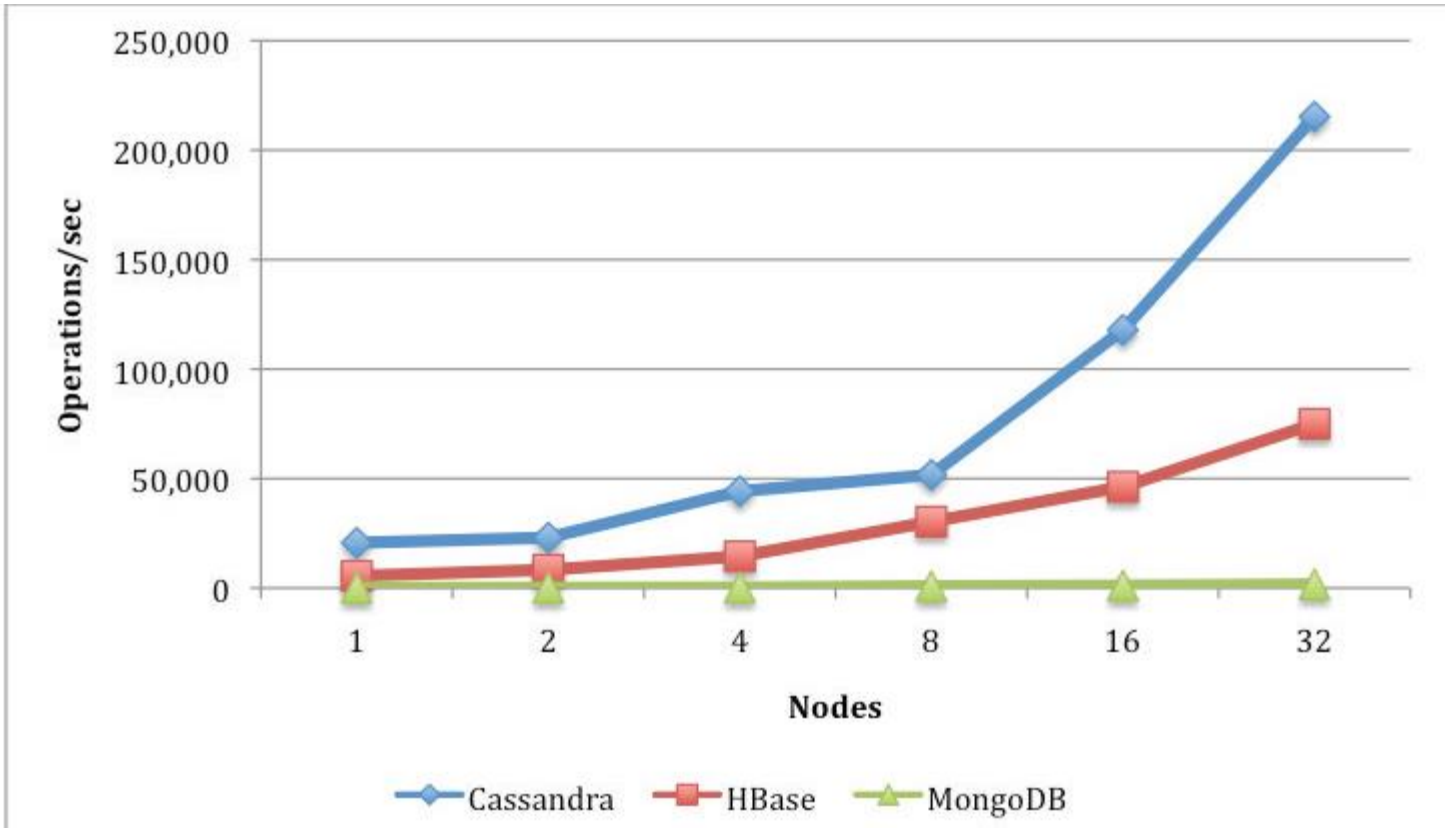
- Different workloads are important for different use cases and one should really examine the intended usage of a NoSQL database before deciding which product to use.

Read/Write Mix Workload



Complete white paper on this series of tests can be found at:
http://www.datastax.com/resources/whitepapers/benchmarking-top-nosql-databases

Write-mostly Workload



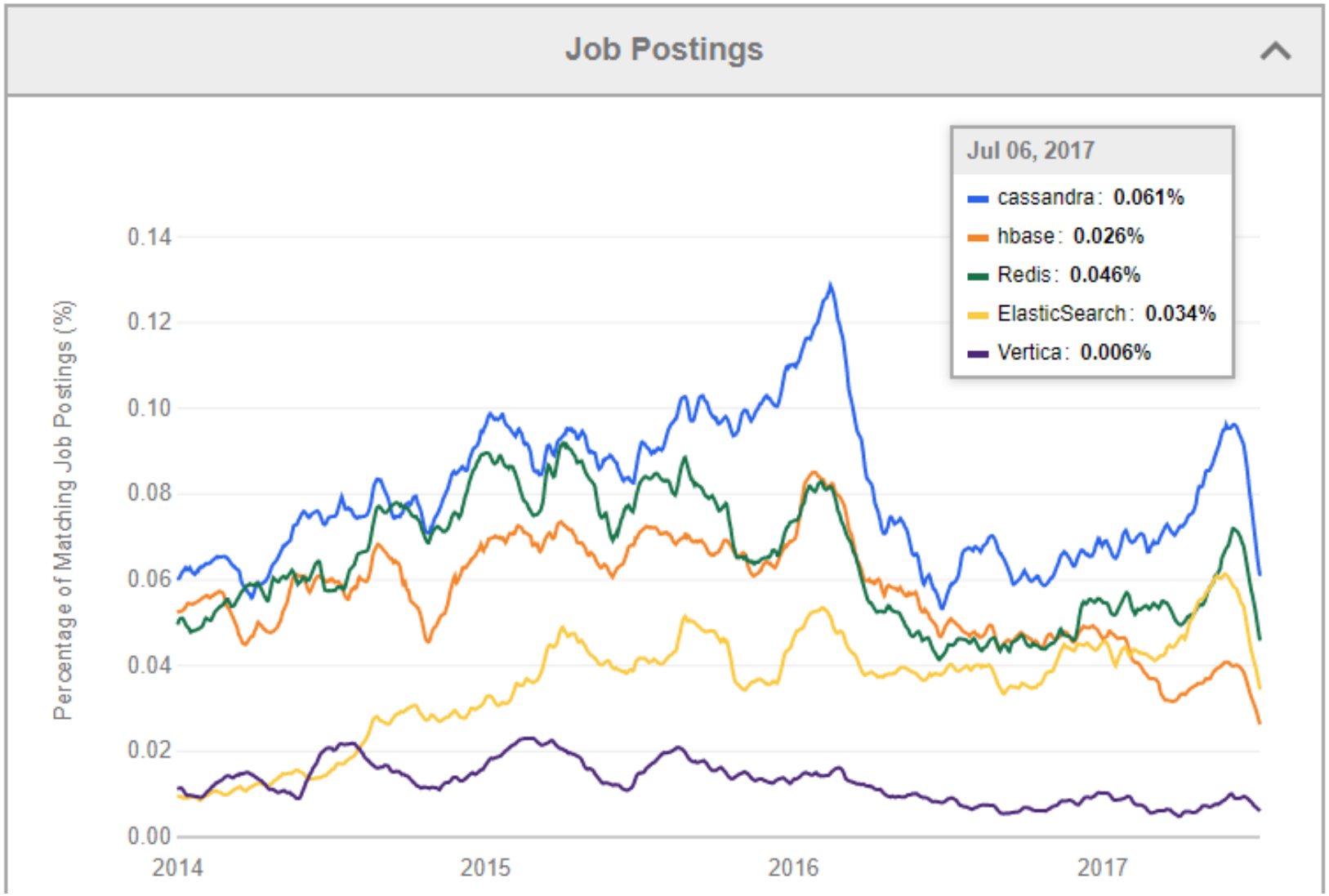
- Here we presented only 3 of 8 tested workload scenarios. It appears that in all of them Cassandra demonstrated the highest throughput.

Choose

- Based on results for all tested workloads, it appears that Cassandra is an overwhelming winner.
- Cassandra has some “deficiencies”. For example, it does not implement a very strong consistency model and does not consider ACID properties to be sacrosanct.
- ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. Old fashioned RDBMS-s like Oracle, DB2, PostgreSQL, MySQL, and others guaranty that those properties will hold in any of your transactions.
- HBase, for example, pays due respect to consistency property and you might decide to select HBase over Cassandra if data consistency is critical for your application, in spite of HBase’s less-than-stellar performance.
- Also, Cloudera which “owns” HBase claims that they made significant improvements recently. Also, if your data is all in HDFS or you want it in HDFS, HBase is a natural choice.
- In the following we will look at some properties of Cassandra, HBase and MongoDB.

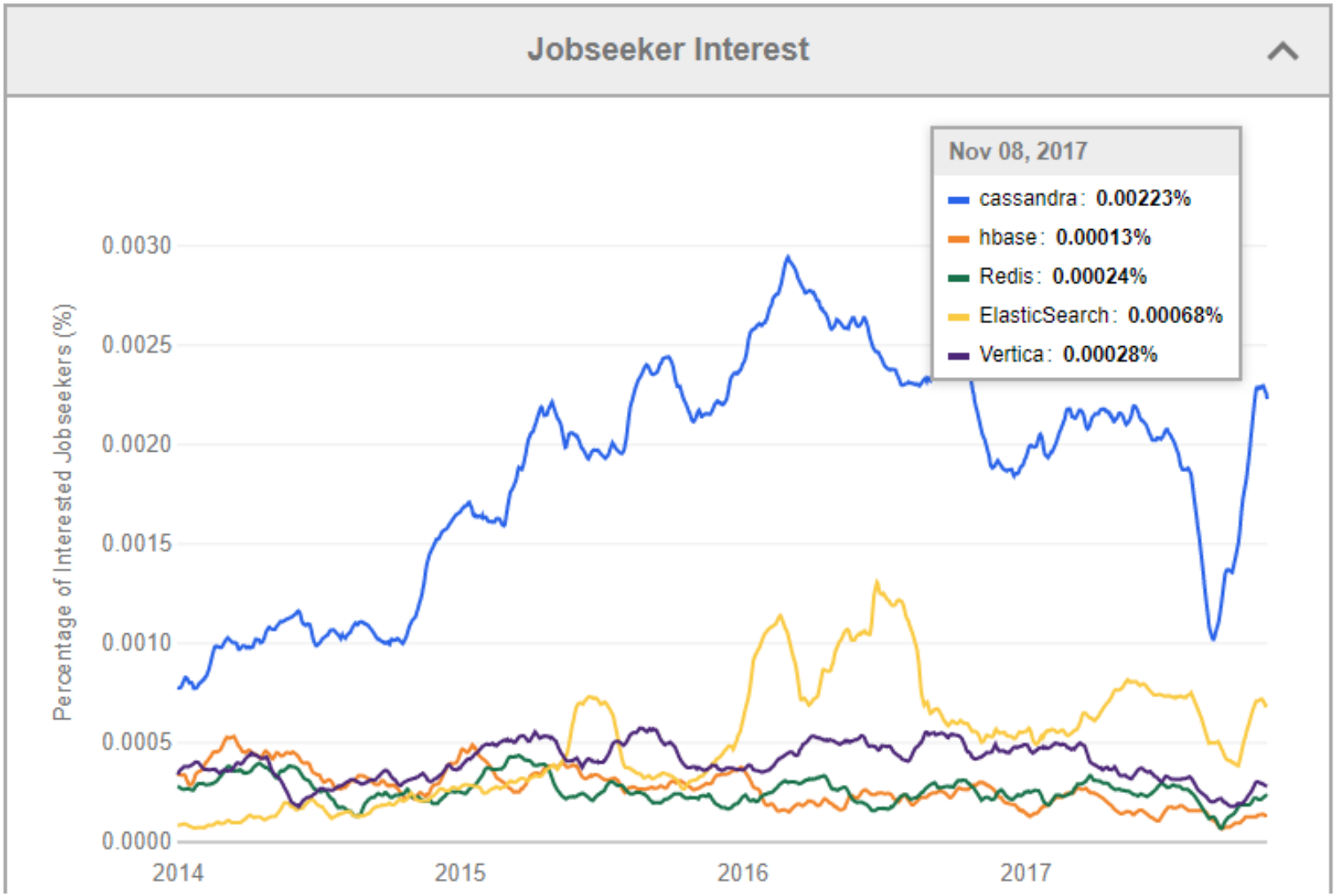
Interest in Various NoSQL Systems

https://www.indeed.com/jobtrends/q-cassandra-q-hbase-q-Redis-q-ElasticSearch-q-Vertica.html



Interest in Various NoSQL Systems

https://www.indeed.com/jobtrends/q-cassandra-q-hbase-q-Redis-q-ElasticSearch-q-Vertica.html



Azure Cosmos DB

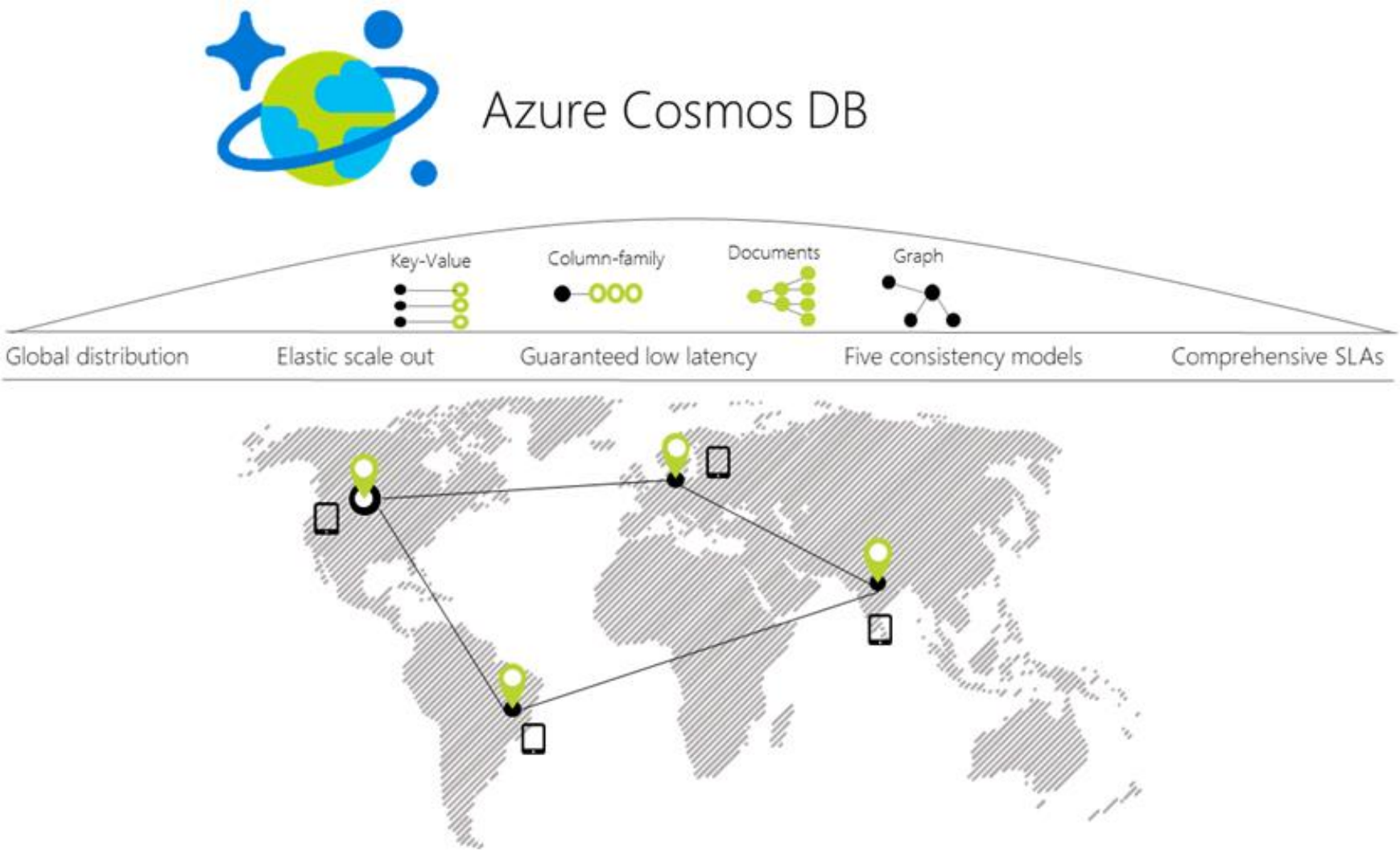
Azure Cosmos DB

- In Azure one can run any of mentioned NoSQL systems and databases. Some of them come on preconfigured VMs. Azure's user have the freedom to install any version of any NoSQL DB they like using Azure Linux or Windows VMs.
- Microsoft saw that many of databases (relational and NoSQL) are being use together catering to different portions of applications.
- Azure introduces a new service called Azure Cosmos DB, that provides developer with ability to use both standard relational API (SQL) and APIs of most popular NoSQL databases: Cassandra, MongoDB, Azure Tables and soon others.
- Cosmos DB is not just a collection of several NoSQL services.

- Azure Cosmos DB is Microsoft's globally distributed, multi-model database.
- Azure Cosmos DB enables you to elastically and independently scale throughput and storage across any number of Azure's geographic regions.
- It offers throughput, latency, availability, and consistency guarantees with comprehensive service level agreements (SLAs), "something no other database service can offer".
- Cosmos DB is an example of emerging **Database as a Service (DaaS)** pattern.
- Oracle is building similar offerings. AWS already has several DaaS types. Google's BigTable is a DaaS. Eventually, Cosmos DB will be imitated by all.

Global Database

- Replicas of Azure Cosmos DB could reside in any (many) Azure regions.



SLA

OPERATION	MAXIMUM UPPER BOUND ON PROCESSING LATENCY
All Database Account configuration operations	2 Minutes
Add a new Region	60 Minutes
Manual Failover	5 Minutes
Resource Operations	5 Sec
Media Operations	60 Sec
MONTHLY AVAILABILITY PERCENTAGE	**SERVICE CREDIT**
< 99.99%	10%
< 99%	25%
MONTHLY READ AVAILABILITY PERCENTAGE	**SERVICE CREDIT**
< 99.999%	10%
<99%	25%

Key Capabilities

- **Turnkey global distribution**
  - Data in any number of Azure regions. This enables you to put your data where your users are, ensuring the lowest possible latency to your customers.
  - Using Azure Cosmos DB's multi-homing APIs, the app always knows where the nearest region is and sends requests to the nearest data center. All of this is possible with no config changes. You set your write-region and as many read-regions as you want, and the rest is handled for you.
- **Multiple data models and popular APIs for accessing and querying data**
  - The atom-record-sequence (ARS) based data model that Azure Cosmos DB is built on natively supports multiple data models, including but not limited to document, graph, key-value, table, and columnar data models.
  - APIs for the following data models are supported with SDKs available in multiple languages:
    - SQL API: A schema-less JSON database engine with SQL querying capabilities.
    - MongoDB API: A MongoDB database service built on top of Cosmos DB. Compatible with existing MongoDB libraries, drivers, tools and applications.
    - Table API: A key-value database service built to provide premium capabilities for Azure Table storage applications.
    - Graph (Gremlin) API: A graph database service built following the Apache TinkerPop specification.
    - Cassandra API: A key/value store built on the Apache Cassandra implementation.

Key Capabilities

- **Elastically scale throughput and storage on demand, worldwide**
  - Easily scale database throughput at a per-second granularity, and change it anytime you want.
  - Scale storage size transparently and automatically to handle your size requirements now and forever.
- **Low cost of ownership**
  - Five to 10 times more cost effective than a non-managed solution.
  - Three times cheaper than DynamoDB.
- **Build highly responsive and mission-critical applications**
  - Azure Cosmos DB guarantees end-to-end low latency at the 99th percentile to its customers.
  - For a typical 1-KB item, Cosmos DB guarantees end-to-end latency of reads under 10 ms and indexed writes under 15 ms at the 99th percentile, within the same Azure region. The median latencies are significantly lower (under 5 ms).

Key Capabilities

- **Ensure "always on" availability**
  - 99.99% availability SLA for all single region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability on all multi-region database accounts.
  - Deploy to any number of Azure regions for higher availability.
  - Simulate a failure of one or more regions with zero-data loss guarantees.
- **Write globally distributed applications, the right way**
  - Five consistency models provide a spectrum of strong SQL-like consistency all the way to NoSQL-like eventual consistency, and every thing in between.
- **Money back guarantees**
  - Your data gets there fast, or your money back.
  - Service level agreements for availability, latency, throughput, and consistency.
- **No database schema/index management**
  - Stop worrying about keeping your database schema and indexes in-sync with your application's schema. We're schema-free.
  - Azure Cosmos DB's database engine is fully schema-agnostic – it automatically indexes all the data it ingests without requiring any schema or indexes and serves blazing fast queries.

Tunable Data Consistency Levels

- Azure Cosmos DB is designed from the ground up with global distribution in mind for every data model. It is designed to offer predictable low latency guarantees and multiple well-defined relaxed consistency models. Currently, Azure Cosmos DB provides five consistency levels: strong, bounded-staleness, session, consistent prefix, and eventual.

Consistency Level	Guarantees
Strong	Linearizability. Reads are guaranteed to return the most recent version of an item.
Bounded Staleness	Consistent Prefix. Reads lag behind writes by k prefixes or t interval
Session	Consistent Prefix. Monotonic reads, monotonic writes, read-your-writes, write-follows-reads
Consistent Prefix	Updates returned are some prefix of all the updates, with no gaps
Eventual	Out of order reads

Relational, NoSQL, Cosmos DB

- Cosmos DB is more than a collection of APIs

Capabilities	Relational databases	NoSQL databases	Azure Cosmos DB
Global distribution	No	No	Yes, turnkey distribution in 30+ regions, with multi-homing APIs
Horizontal scale	No	Yes	Yes, you can independently scale storage and throughput
Latency guarantees	No	Yes	Yes, 99% of reads in <10 ms and writes in <15 ms
High availability	No	Yes	Yes, Cosmos DB is always on, has PACELC tradeoffs, and provides automatic & manual failover options
Data model + API	Relational + SQL	Multi-model + OSS API	Multi-model + SQL + OSS API (more coming soon)
SLAs	Yes	No	Yes, comprehensive SLAs for latency, throughput, consistency, availability

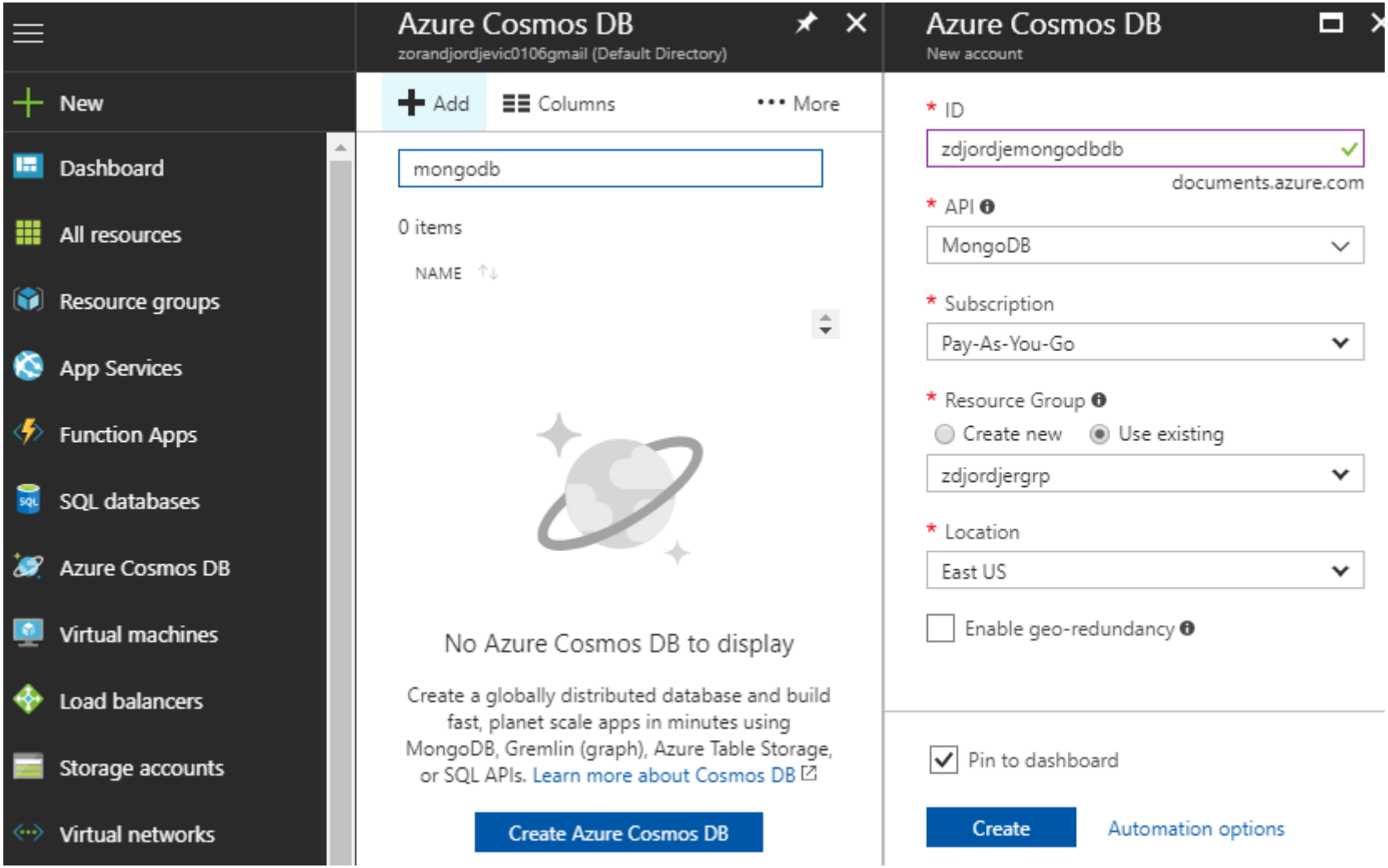
Who Needs Cosmos DB

- Any web, mobile, gaming, and IoT application that need to handle massive amounts of reads and writes on a global scale with low response times for a variety of data.
- Such applications will benefit from Azure Cosmos DB's guaranteed availability, high throughput, low latency, and tunable consistency.

- Note: It appears that every API (MongoDB, Cassandra, SQL, Graph) requires a separate database creation.
- Note: Cassandra API is not available to general public yet.

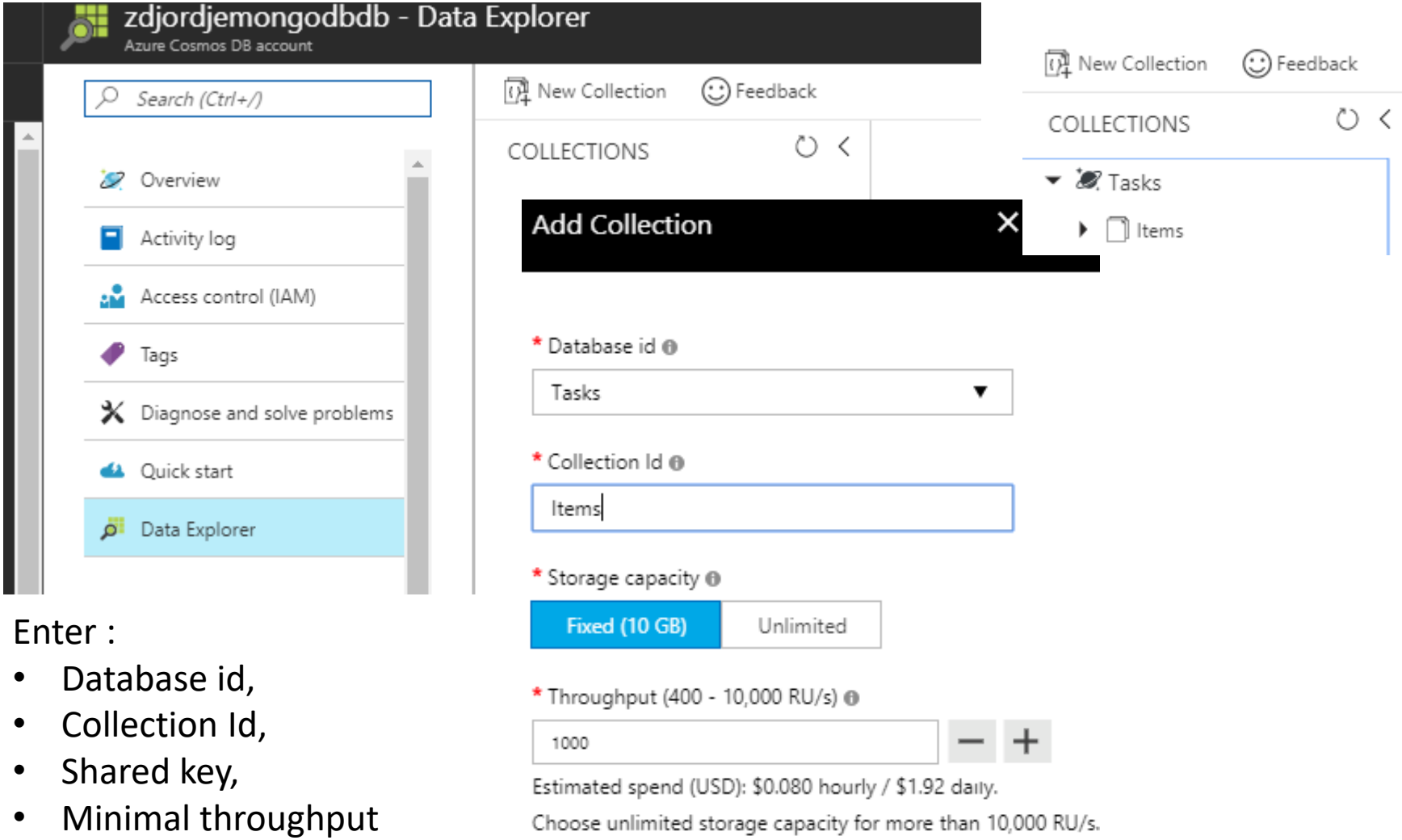
Create MongoDB Database Account

- In Poral, select Azure Cosmos DB > + Add > API > MongoDB



MongoDB Account is Created

- Select Data Explorer, +New Collection. Add Collection opens on far right



Enter :

- Database id,
- Collection Id,
- Shared key,
- Minimal throughput

Hit OK

New collection Items shows up

Clone the Sample Java Application

- On Windows or Linux command prompt run the following Git command:

git clone https://github.com/Azure-Samples/azure-cosmos-db-mongodb-java-getting-started.git

- The command will create a local copy of Java MongoDB client application.
- Go to Eclipse and do: `File > Import > Existing Maven Project` named `GetStarted`. Change the name of your project.
- Maven `pom.xml` file contains one interesting dependency:

```
<!-- https://mvnrepository.com/artifact/org.mongodb/mongo-java-driver -->
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.4.2</version>
</dependency>
```

- That is MongoDB "original" Java driver. Your existing MongoDB projects could connect to this database.

Client Code: `Program.java`

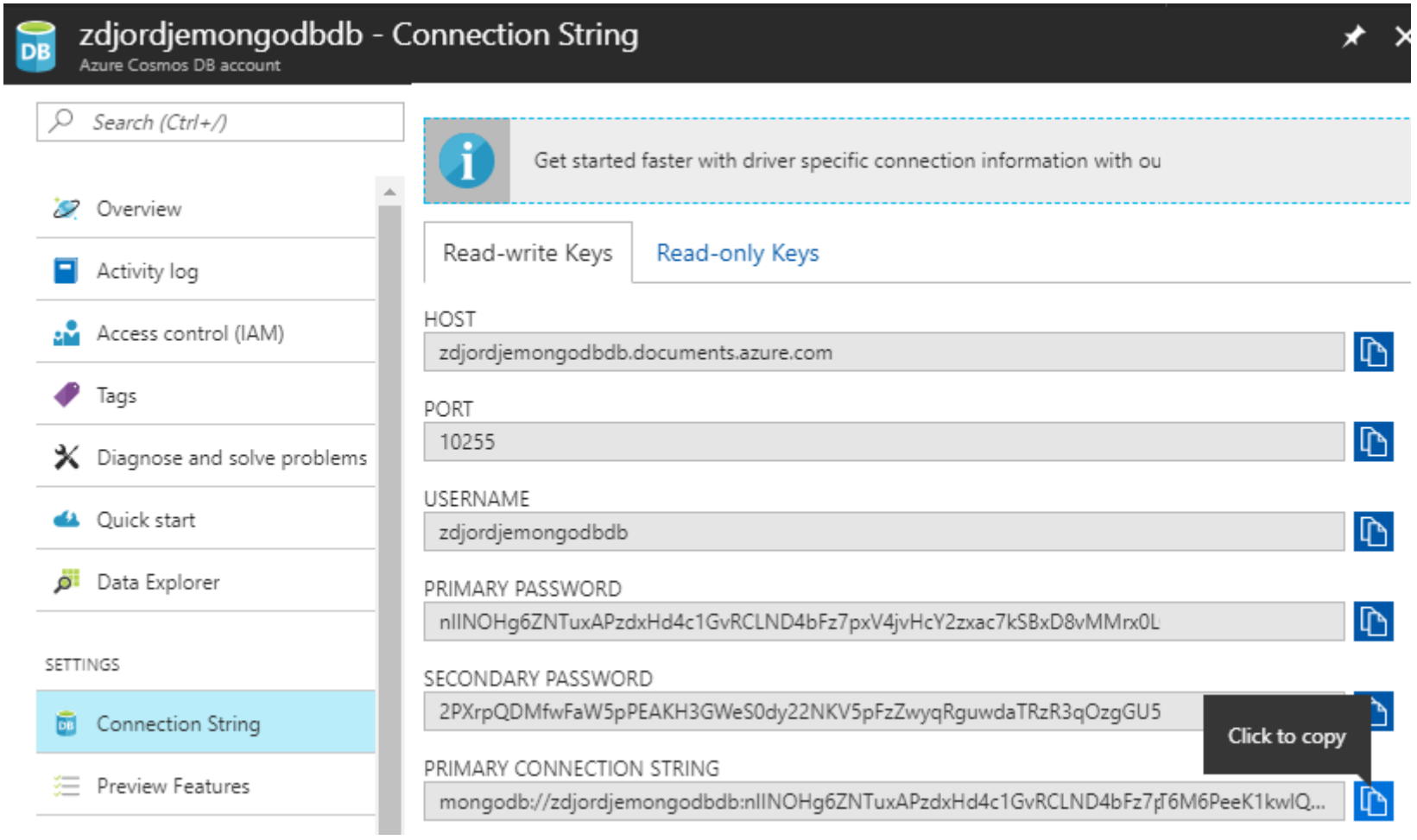
```
package GetStarted;

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoClientURI;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;

/**
 * Simple application, uses Azure Cosmos DB with the MongoDB API and Java.
 */
public class Program {
    public static void main(String[] args)
    {
/*
* Replace connection string from the Azure Cosmos Portal
        */
        MongoClientURI uri = new
MongoClientURI("mongodb://zdjordjecosmosdb:ny0CEpHN5k5LF0JkACVos4rVHFzPFxtm
6vCP17WgP85idFa94e8oZnOJ6v5uSCYcViW7xFK0pxx0HiAhlqrBGA==@zdjordjecosmosdb.d
ocuments.azure.com:10255/?ssl=true&replicaSet=globaldb");

        MongoClient mongoClient = null;
        try {
            mongoClient = new MongoClient(uri);
```

Client Code: `Program.java`

```
            // Get database
            MongoDatabase database = mongoClient.getDatabase("db");

            // Get collection
            MongoCollection<Document> collection = database.getCollection("coll");

            // Insert documents
            Document document1 = new Document("fruit", "apple");
            collection.insertOne(document1);

            Document document2 = new Document("fruit", "mango");
            collection.insertOne(document2);

            // Find fruits by name
            Document queryResult = collection.find(Filters.eq("fruit",
"apple")).first();
            System.out.println(queryResult.toJson());

            System.out.println( "Completed successfully" );

        } finally {
        if (mongoClient != null) {
        mongoClient.close();
        }
        }
    }
}
```

Connection String

- Client Program.java requires the connection string. On you MongoDB blade, select Connection String and copy Primary Connection String. Pass the string to `MongoClientURI()`

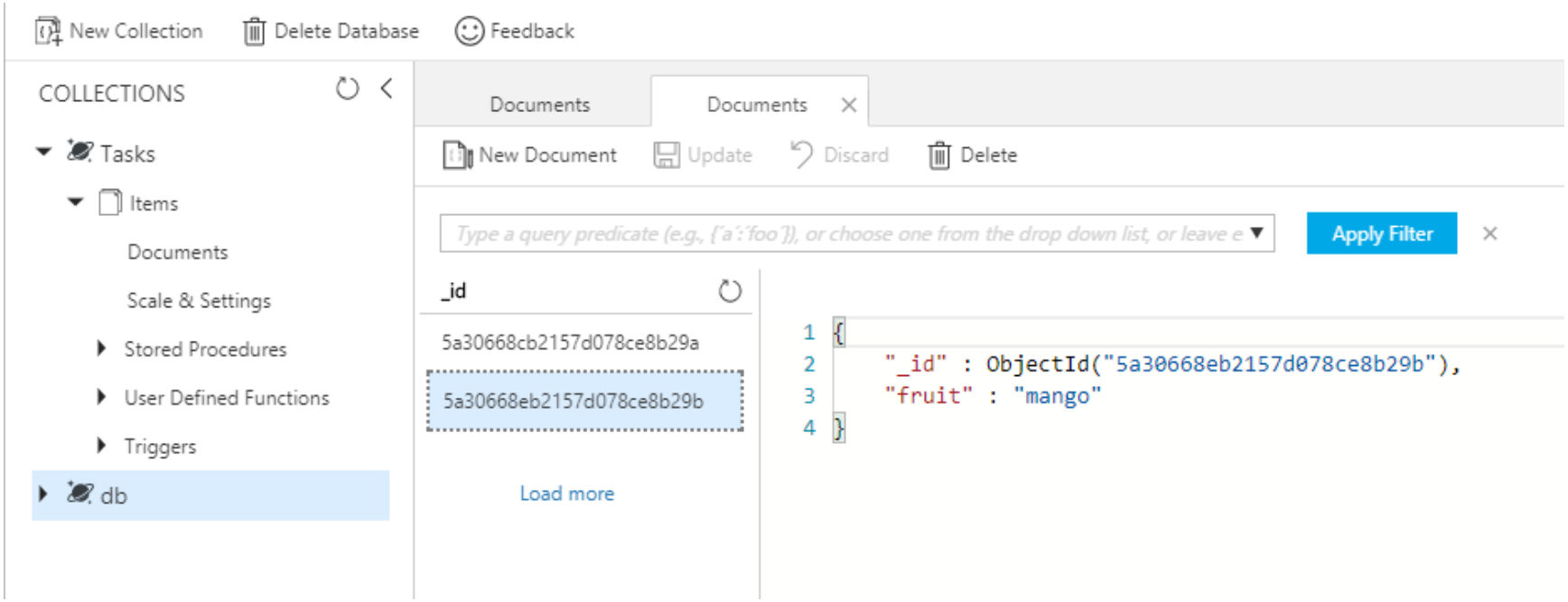


Run Client

- Run Program.java as a Java Application

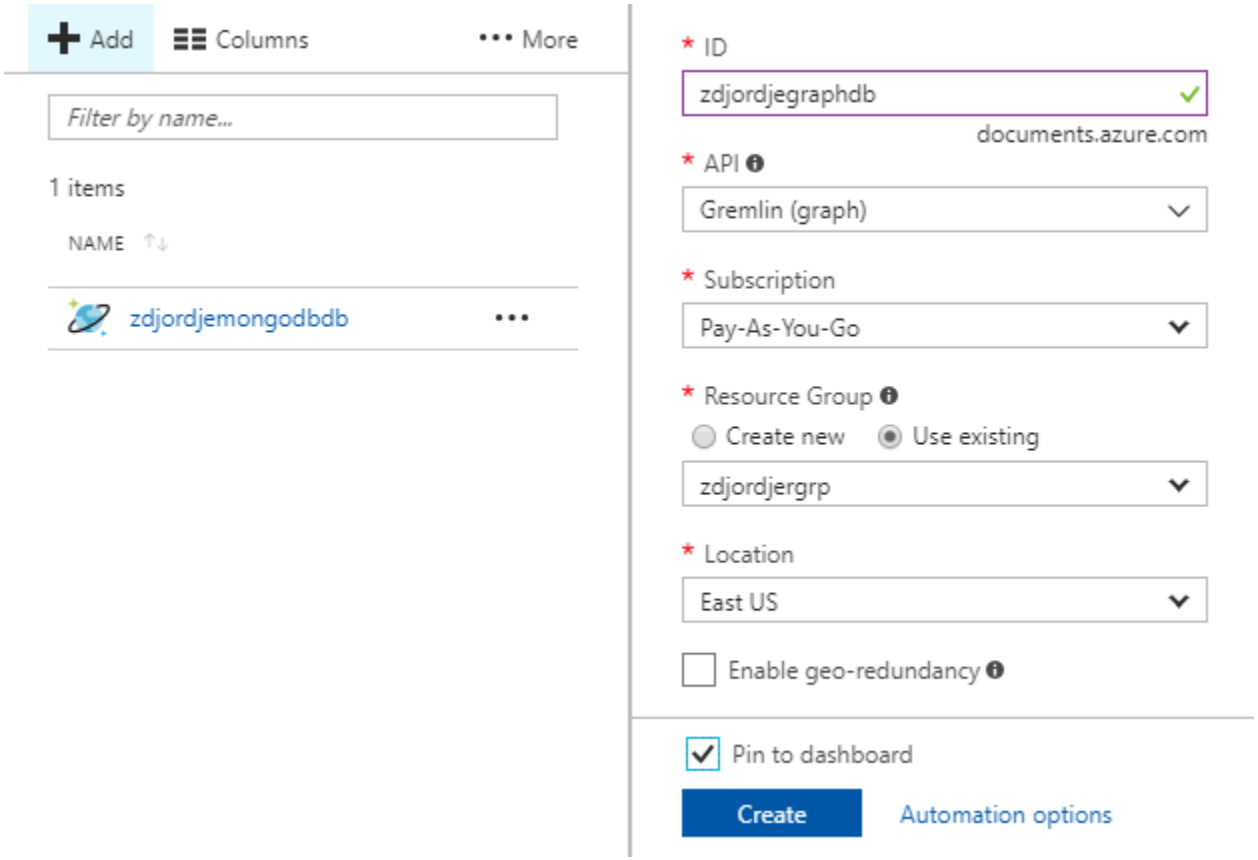
```
Dec 12, 2017 6:30:20 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Cluster created with settings
{hosts=[zdjordjemongodbdb.documents.azure.com:10255], mode=MULTIPLE,
requiredClusterType=REPLICA_SET, Dec 12, 2017 6:30:21 PM
com.mongodb.diagnostics.logging.JULLogger log
INFO: Opened connection [connectionId{localValue:3, serverValue:12575124}] to
bl4prdddc02-docdb-1.documents.azure.com:10255
{ "_id" : { "$oid" : "5a30668cb2157d078ce8b29a" }, "fruit" : "apple" }
Completed successfully
```

Data Explorer View



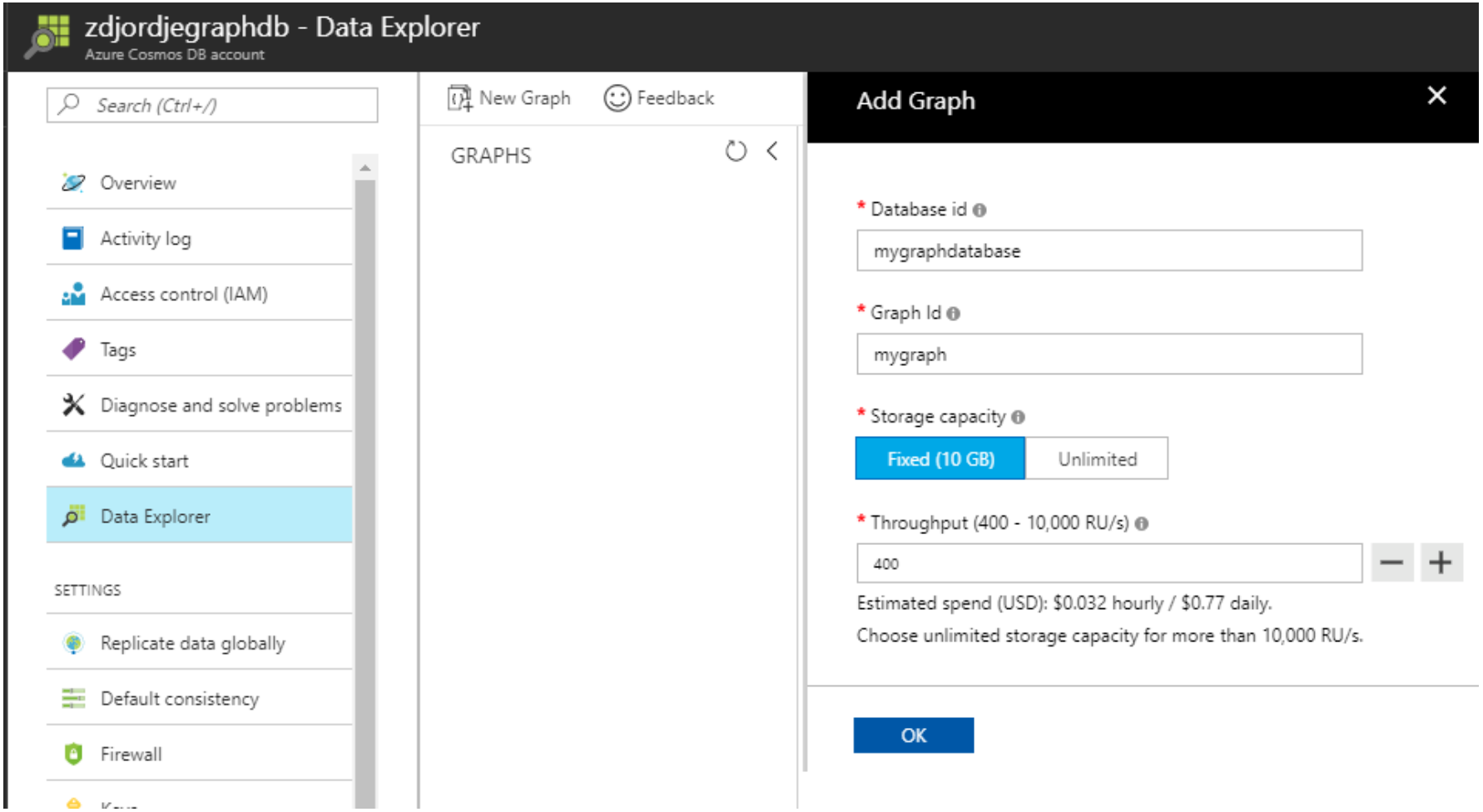
Create GraphDB Account

- Process for creation of a Graph DB account is the same, except that you select Gremlin (graph) API:



Data Explorer, Add a Graph

- Hit Data Explorer, New Graph and fill in the fields in Add Graph:



Git Clone Sample App

- From Linux or Windows prompt do:

git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-java-getting-started.git

- In Eclipse, go to File > Import > Existing Maven Project.
- That is it.
- This time you have fix configuration file: remote.yaml

hosts: [zdjordjegraphdb.graphs.azure.com]

port: 443

username: /dbs/graphdb/colls/sample-graph

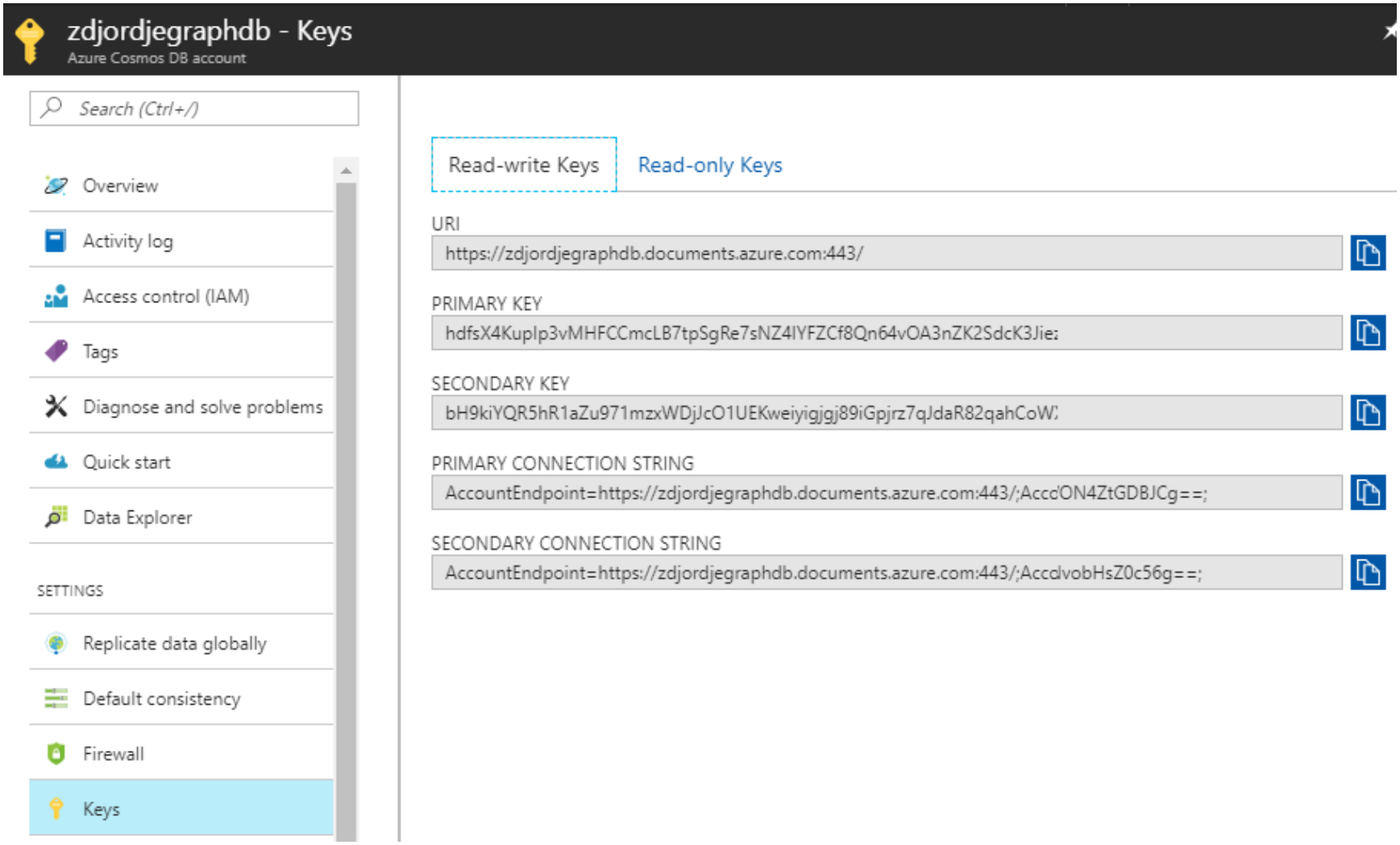
password: 4MVox7NzVWp1B0bfhGCRHNZaox1zUMlBqmyiSKBzQ2b5hUybFP8BH9mVs4FLQcw7rLusyrEZpk32NIt3iQP4GQ==

connectionPool: {

enableSsl: true}

serializer: { className: org.apache.tinkerpop.gremlin.driver.ser.GraphSONMessageSerializerV1d0, config: { serializeResultToString: true }}

On Graph DB blade, Select Keys



Update your connection information

- In the Azure portal, click Keys.
- Copy the first portion of the URI value.
- View and copy an access key in the Azure portal, Keys page
- Open the src/remote.yaml file and paste the value over $name$ in hosts: [$name$.graphs.azure.com].

- Line 1 of remote.yaml should now look similar to

- hosts: [test-graph.graphs.azure.com]

- In the Azure portal, use the copy button to copy the PRIMARY KEY and paste it over $masterKey$ in password: $masterKey$.

- Line 4 of remote.yaml should now look similar to

pom.xml

- I could not make sample Graph project work without adding following dependencies to `pom.xml` file:

```
<dependency>
      <groupId>org.apache.tinkerpop</groupId>
      <artifactId>gremlin-driver</artifactId>
      <version>3.2.4</version>
    </dependency>
         <dependency>
           <groupId>org.slf4j</groupId>
           <artifactId>slf4j-api</artifactId>
           <version>1.6.6</version>
       </dependency>
       <dependency>
           <groupId>org.slf4j</groupId>
           <artifactId>slf4j-simple</artifactId>
           <version>1.6.6</version>
       </dependency>
```

Appendix: MongoDB

What is MongoDB?

MongoDB (from "humongous") is an

- Open-source document database
- Written in C++
- Agile and scalable.

History of MongoDB

- First developed by 10gen (now MongoDB Inc.) in October 2007 as a component of a planned platform as a service product, the company shifted to an open source development model in 2009, with 10gen offering commercial support and other services.
- Since then, MongoDB has been adopted as backend software by a number of major websites and services, including Craigslist, eBay, Foursquare, SourceForge, and The New York Times, among others. MongoDB is the most popular NoSQL database system.

- BSON is a computer data interchange format used mainly as a data storage and network transfer format in the MongoDB database.
- It is a binary form for representing simple data structures and associative arrays (called objects or documents in MongoDB).
- The name "BSON" is based on the term JSON and stands for "Binary JSON".

Example JSON Schema:

```
{
    "name": "Product",
    "properties": {
        "id": {
            "type": "number",
            "description": "Product identifier",
            "required": true
        },
        "name": {
            "type": "string",
            "description": "Name of the product",
            "required": true
        },
        "price": {
            "type": "number",
            "minimum": 0,
            "required": true
        },
        "tags": {
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "stock": {
            "type": "object",
            "properties": {
                "warehouse": {
                    "type": "number"
                },
                "retail": {
                    "type": "number"
                }
            }
        }
    }
}
```

Terminology and Concepts

SQL Terms/Concepts	MongoDB Terms/Concepts
database	*database*
table	*collection*
row	*document* or *BSON* document
column	*field*
index	*index*
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	*primary key* In MongoDB, the primary key is automatically set to the *_id* field.

SQL to MongoDB Mapping

SQL Schema Statements	MongoDB Schema Statements
CREATE TABLE users (id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id))	db.users.insert({ user_id: "abc123", age: 55, status: "A" }) Implicitly created on first insert() operation. The primary key _id is automatically added if _id field is not specified.
SELECT * FROM users	db.users.find()

MongoDB Server-Side JavaScript

- JavaScript may be executed in the MongoDB server processes for various functions, such as query enhancement and map/reduce processing.
- Example:

```
for (var i = 1; i <= 25; i++) db.testData.insert( { x : i } )
```

```
db.testData.find()  displays first 20 docs in the collection
```

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be6"), "x" : 1 } { "_id" :
ObjectId("51a7dc7b2cacf40b79990be7"), "x" : 2 } { "_id" :
ObjectId("51a7dc7b2cacf40b79990be8"), "x" : 3 }
```

Data Models

- Data in MongoDB has a *flexible schema*.
- *Collections* do not enforce *document* structure.
- This flexibility gives you data-modeling choices to match your application and its performance requirements.
- In other words: Data Modeling for MongoDB Applications documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold **different types** of data.

Indexes

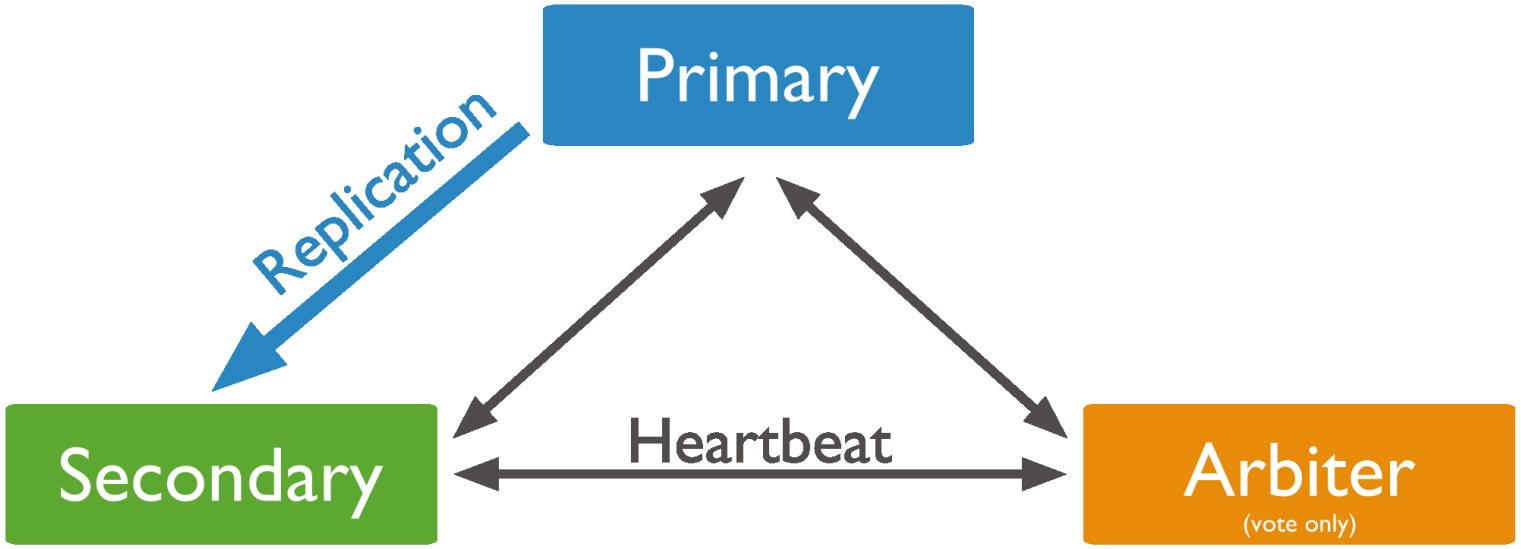
- Indexes provide high performance read operations for frequently used queries.
- Full Index Support - Index on any attribute, just like you're used to.
- Example, **Create an Index on a Single Field:**

```
db.people.ensureIndex( { "phone-number": 1 } )
```

- A value of 1 specifies an index that orders items in ascending order.
- A value of -1 specifies an index that orders items in descending order

Replication

- Replication provides redundancy and increases data availability. Example:



- The primary is the only member in the replica set that receives write operations. A secondary maintains a copy of the primary's data set. Replica sets may have arbiters to add a vote in elections of for primary.

Sharding

- Sharding is the process of storing data records across multiple machines and is MongoDB's approach to meeting the demands of data growth.
- As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput.
- Sharding solves the problem with horizontal scaling.
- With sharding, you add more machines to support data growth and the demands of read and write operations.

Java Driver API

```
MongoClient mongoClient = new MongoClient();
mongoClient.close();
DB db = mongoClient.getDB(dbName);
DBCollection dBCollection = db.getCollection(c);
BasicDBObject doc = new BasicDBObject();
doc.put(string1, string2);
newCollection.insert(doc);
collection.ensureIndex(new BasicDBObject("STR", 1),
indexName);
BasicDBObject query = new BasicDBObject();
query.put("_id", new ObjectId(id));
DBObject dbObj = dbCollection.findOne(query);
```