JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

Master of Computer Applications

18MCA301 – **NoSQL Databases**

Module -4

Key-Value Based Databases

III General

Dr Gangothri

September 2020

- A key-value store is a simple database that when presented with a simple string (the key) returns an arbitrary large BLOB of data (the value).
- Key-value stores have no query language; they provide a way to add and remove key-value pairs (a combination of key and value where the key is bound to the value until a new value is assigned) into/from a database.
- The key in a key-value store is flexible and can be represented by many formats:

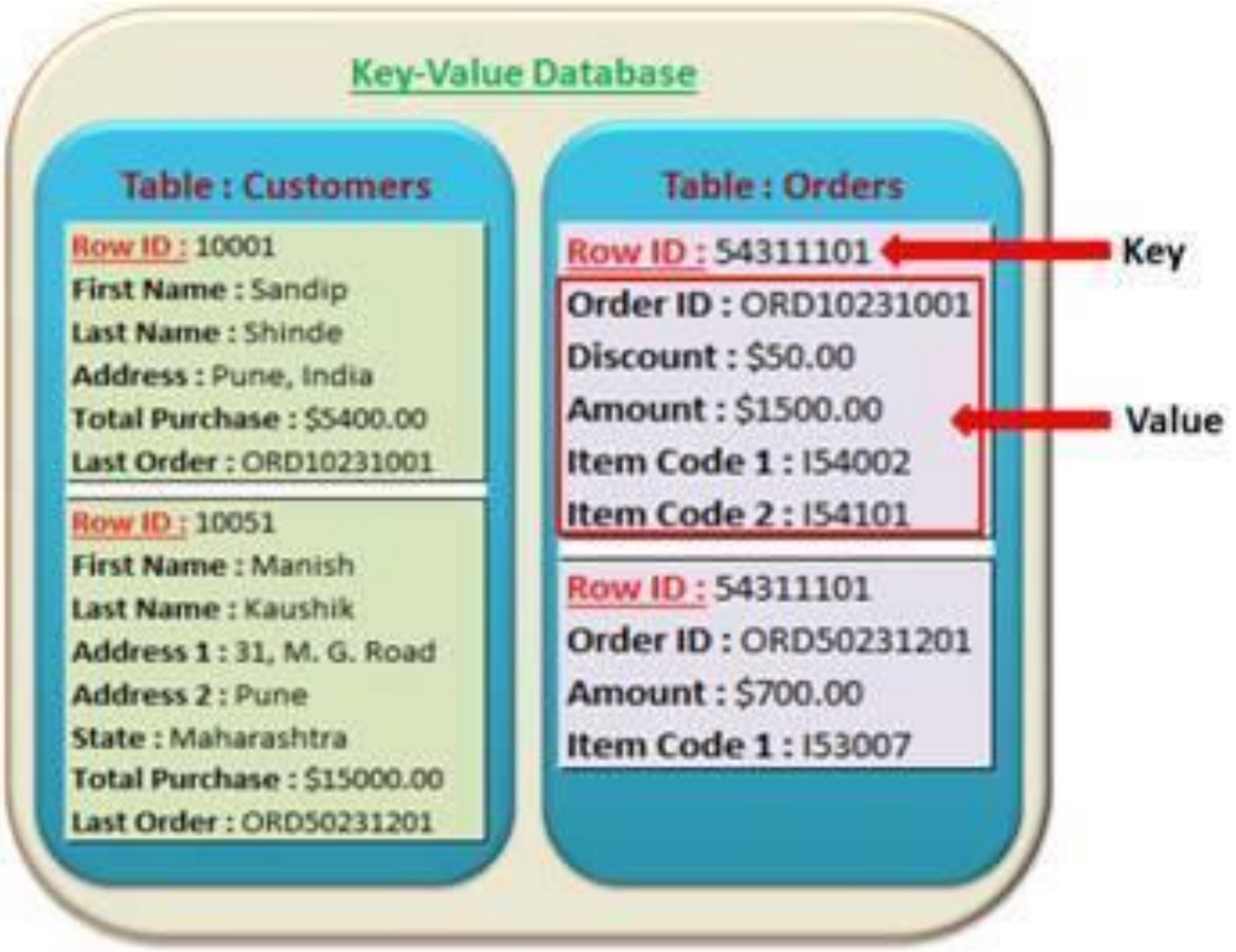
 Logical path names to images or files

 Artificially generated strings created from a hash of the value

 REST web service calls

 SQL queries

- Values, like keys, are also flexible and can be any BLOB of data, such as images, web pages, documents, or videos.


Key-Value Database
Table : Customers
Row ID : 10001
First Name : Sandip
Last Name : Shinde
Address : Pune, India
Total Purchase : $5400.00
Last Order : ORD10231001
Row ID : 10051
First Name : Manish
Last Name : Kaushik
Address 1 : 31, M. G. Road
Address 2 : Pune
State : Maharashtra
Total Purchase : $15000.00
Last Order : ORD50231201
Table : Orders
Row ID : 54311101
Order ID : ORD10231001
Discount : $50.00
Amount : $1500.00
Item Code 1 : I54002
Item Code 2 : I54101
Row ID : 54311101
Order ID : ORD50231201
Amount : $700.00
Item Code 1 : I53007
Key
Value

Benefits of using a key-value store

1. Precision service levels
2. Precision service monitoring and notification
3. Scalability and reliability
4. Portability and lower operational costs

SCHOOL OF COMPUTER SCIENCE AND IT

Precision service levels

- When you have a simple data service interface that's used across multiple applications, you can focus on things like creating precise service levels for data services.
- A service level doesn't change the API; it only puts precise specifications on how quickly or reliably the service will perform under various load conditions.
- For example, for any data service you might specify
  - □ The maximum read time to return a value
  - □ The maximum write time to store a new key-value pair
  - □ How many reads per second the service must support
  - □ How many writes per second the service must support
  - □ How many duplicate copies of the data should be created for enhanced reliability
  - □ Whether the data should be duplicated across multiple geographic regions if some data centers experience failures
  - □ Whether to use transaction guarantees for consistency of data or whether eventual consistency is adequate

Precision Service Monitoring And Notification

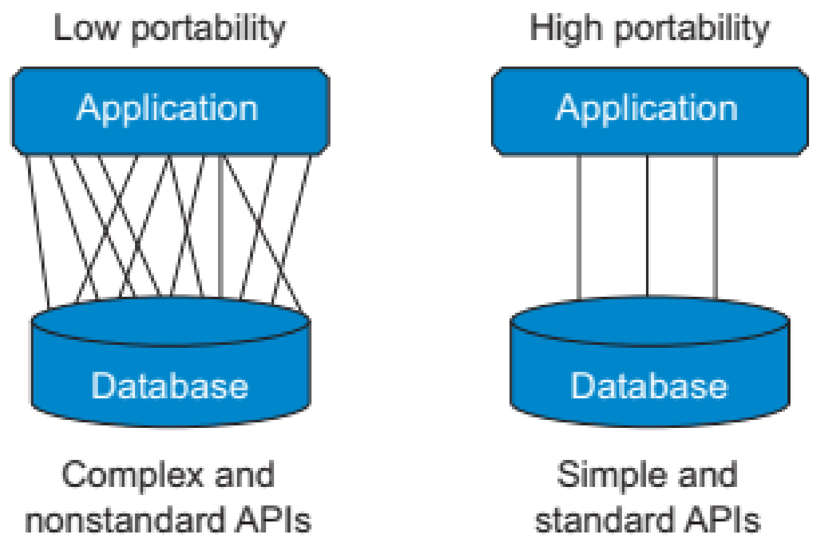
- In addition to specifying service levels, you can also invest in tools to monitor your service level.
- When you configure the number of reads per second a service performs, setting the parameter too low may mean the user would experience a delay during peak times.
- By using a simple API, detailed reports showing the expected versus actual loads can point you to system bottlenecks that may need additional resource adjustments.
- Automatic notification systems can also trigger email messages when the volume of reads or writes exceeds a threshold within a specified period of time.

Scalability And Reliability

- When a database interface is simple, the resulting systems can have higher scalability and reliability.
- This means you can tune any solution to the desired requirements.
- Keeping an interface simple allows novice as well as advanced data modelers to build systems that utilize this power.
- Your only responsibility is to understand how to put this power to work solving business problems.
- A simple interface allows you to focus on load and stress testing and monitoring of service levels.
- Because a key-value store is simple to set up, you can spend more time looking at how long it takes to put or get 10,000 items.
- It also allows you to share these load- and stress-testing tools with other members of your development team.

Portability And Lower Operational Costs

- One of the challenges for information systems managers is to continually look for ways to lower their operational costs of deploying systems.
- It's unlikely that a single vendor or solution will have the lowest cost for all of your business problems.
- Ideally, information systems managers would like to annually request data service bids from their database vendors.
- In the traditional relational database world, this is impractical since porting applications between systems is too expensive compared to the relative savings of hosting your data on a new vendor's system.
- The more complicated and non-standardized they are, the less portable they can be and the more difficult moving them to the lowest cost operator is.



Managing Availability

- Key-value stores typically provide a wide range of consistency and durability models — that is, between availability and partition tolerance and between consistent and partition tolerance.
- **Trading consistency:**
  - Key-value stores typically trade consistency in the data in order to improve write times.
  - Voldemort, Riak, and Oracle NoSQL are all eventually consistent key-value stores.
  - They use a method called read repair. Here are the two steps involved in read repair:

  1. At the time of reading a record, determine which of several available values for a key is the latest and most valid one.
  2. If the most recent value can’t be decided, then the database client is presented with all value options and is left to decide for itself.

  Good examples for using eventually consistent key-value stores include sending social media posts and delivering advertisements to targeted users.

  If a tweet arrives late or a five-minute-old advertisement is shown, there’s no catastrophic loss of data.

- **Implementing ACID support:**
  - Aerospike and Redis are notable exceptions to eventual consistency.
  - Both use shared-nothing clusters, which means each key has the following:

  A master node: Only the masters provide answers for a single key, which ensures that you have the latest copy.

  Multiple slave replica nodes: These contain copies of all data on a master node. Aerospike provides full ACID transactional consistency by allowing modifications to be flushed immediately to disk before the transaction is flagged as complete to the database client.

  - Here are some examples of when you may need an ACID-compliant key-value store:

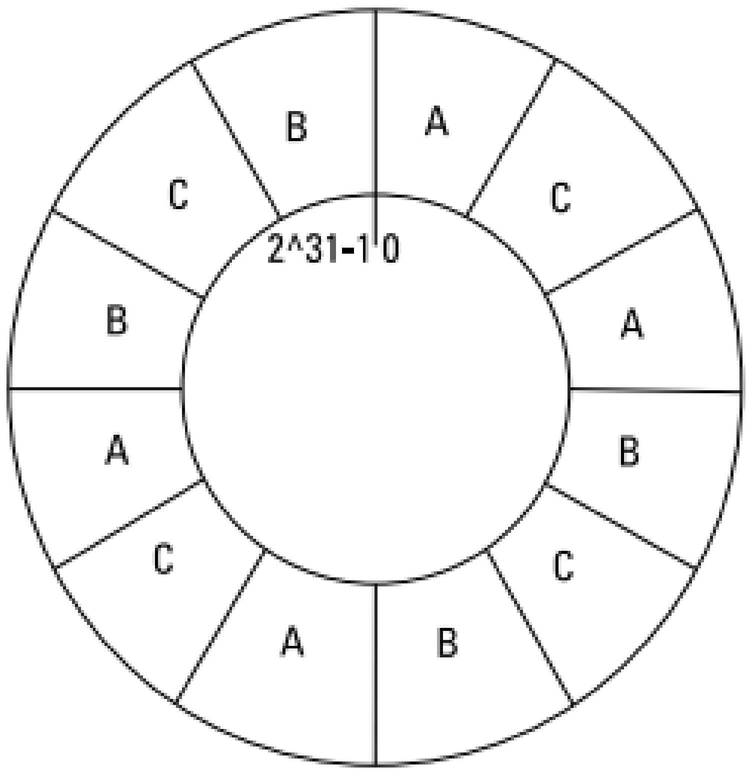
  1. When receiving sensor data that you need for an experiment.
  2. In a messaging system where you must guarantee receipt.

  - Redis, for example, provides a Publish/Subscribe mechanism that acts as a messaging server back end. This feature combined with ACID support allows for durable messaging.

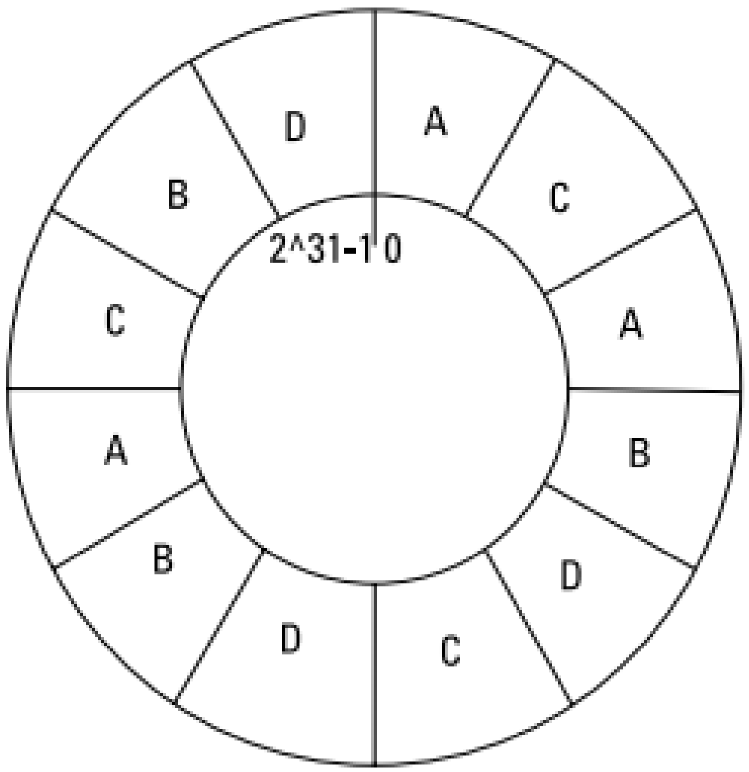
Managing Keys

- Key-value stores' fast read capabilities stem from their use of well-defined keys.
- These keys are typically hashed, which gives a key-value store a very predictable way of determining which partition (and thus server) data resides on.
- A particular server manages one or more partitions.
- A good key enables you to uniquely identify the single record that answers a query without having to look at any values within that record.
- A bad key will require that your application code interprets your record to determine if it does, in fact, match the query.
- **Partitioning:**
  - Partition design is important because some key-value stores, such as Oracle NoSQL, do not allow the number of partitions to be modified once a cluster is created.
  - Their distribution across servers, though, can be modified. So start with a large number of partitions that you can spread out in the future. One example of partitioning is Voldemort's consistent hashing approach

- The same partitions spread across three servers initially and then across four servers later. The number of partitions stays the same, but their allocation is different across servers.
- The same is true of their replicas.



12 partitions, 3 nodes



12 partitions, 4 nodes

- **Accessing data on partitions:**
  - Key-value stores are highly distributed with no single point of failure.
  - This means there’s no need for a master coordinating node to keep track of servers within a cluster.
  - Cluster management is done automatically by a chat protocol between nodes in the server.
  - Most databases, NoSQL included, pass a request on to all members of a cluster. That cluster either accepts the write internally or passes it one under the hood to the correct node.
  - This setup means an extra network trip between nodes is possible, which canadd to latency.
  - In order to avoid discovery latency, most key-value stores’ client drivers maintain a metadata list of the current nodes in a cluster and which partition key ranges each nod manages.
  - In this way, the client driver can contact the correct server, which makes operations faster.
  - If a new node is added to a cluster and the metadata is out of date, the cluster informs the client driver, which then downloads the latest cluster metadata before resending the request to the correct node.
  - This way maximum throughput is maintained with a minimum of overhead during development. Another side benefit is that there’s no need for a load balancer to pass queries on to the next available, or least-busy, server — only one server ever receives a client request, so there’s no need for load balancing

Managing Data

- **Data types in key-value stores:**
  - Key-value stores typically act as “buckets” for binary data.
  - Some databases do provide strong internal data typing and even schema support.
  - Others simply provide convenient helper functions in their client drivers for serializing common application data structures to a key-value store. Examples include maps, lists, and sorted sets.
  - Oracle NoSQL can operate in two modes:

    →Simple binary store

    →Highly structured Avro schema support
  - An Avro schema is akin to a relational database schema — enforcing a very stringent set of format rules on JavaScript Object Notation (JSON) data stored within the database
  - Other NoSQL databases provide secondary indexes on any arbitrary property of a value that has JSON content. Riak, for example, provides secondary indexes based on document partitioning — basically, a known property within a JSON document is indexed with a type.

- **Replicating data:**
  - Storing multiple copies of the same data in other servers, or even racks of servers, helps to ensure availability of data if one server fails.
  - Server failure happens primarily in the same cluster. You can operate replicas two main ways:

✓**Master-slave**: All reads and writes happen to the master. Slaves take over and receive requests only if the master fails.

  - Master-slave replication is typically used on ACID-compliant key-value stores. To enable maximum consistency, the primary store is written to and all replicas are updated before the transaction completes.
  - This mechanism is called a two-phase commit and creates extra network and processing time on the replicas.

✓**Master-master**: Reads and writes can happen on all nodes managing a key. There's no concept of a "primary" partition owner. Master-master replicas are typically eventually consistent, with the cluster performing an automatic operation to determine the latest value for a key and removing older, stale values.

  - In most key-value stores, this happens slowly — at read time. Riak is the exception here because it has an anti-entropy service checking for consistency during normal operations.

Scaling

- Some aspects of scaling are difficult to pull off at the same time — for example, the ability to handle high-speed data ingestion while simultaneously maximizing the speed of reading.
- Key-value stores are great for high-speed data ingestion. If you have a known, predictable primary key, you can easily store and retrieve data with that key.
- Without complex indexing or query features, key-value stores ensure the maximum ingestion speed of data. The databases needed to accomplish fast storage are relatively simple, but you need to consider hardware such as the following:
- **Memory**: Many databases write to an in-memory storage area first and only checkpoint data to disk every so often.
  - Writing to RAM is fast, so it's a good idea to choose a database that does in-memory writing to provide more throughput.
- **SSD**: High-speed flash storage is great for storing large synchronous writes — for example, large in-memory chunks of new data that are offloaded to disk so that RAM can be reserved for new data.
  - Using high-quality SSDs take advantage of this write speed advantage. Some databases, such as Aerospike, natively support SSD storage to provide maximum throughput.
- **Disk arrays**: It's always better to have more spindles (that is, more discs) with less capacity than one large disk. RAID 10 allows newly written data to be split across many discs, maximizing throughput. It also has the handy benefit of keeping an extra copy of your data in case a hard disk fails.

- **Simple data model – fast retrieval:**
  - Storing all the data you need for an operation against a single key means that if you need the data, you have to perform only one read of the database.
  - Minimizing the number of reads you need to perform a specific task reduces the load on your database cluster and speeds up your application.
  - If you need to retrieve data by its content, use an index bucket.
- **In-memory caching:**
  - Moreover, choosing a database that has an in-memory value cache will improve repeated reading
  - Aerospike is notable for giving you the ability to dynamically reprioritize its use of memory, depending on whether you have a high ingest load or a high query load. If your load varies during the day, you may want to consider using Aerospike.
  - You can also use Redis or a similar key-value store as a secondary layer just for caching.
  - Redis is used frequently in conjunction with NoSQL databases that don’t provide their own high-speed read caching.
  - Having a cache in front of your primary database is generally good practice. If you suffer a distributed denial of service (DDoS) attack, the cache will be hit hard, but the underlying database will carry on as normal.

Reducing Time to Value

- Time to value is the amount of time required from starting an IT project to being able to realize business benefit.
- Key-value stores are the simplest NoSQL databases with regards to data model.
- **Using simple structures:**
  - Key-value stores are more flexible than relational databases in terms of the format of data.
  - Use this flexibility to your advantage to maximize the rate of your application's throughput. For example, if you're storing map tiles, store them in hex format so that they can be rendered immediately in a browser.
  - Data structures change over time.
  - A flexible JSON document is better than a CSV data file or fixed-width data file because JSON structures can easily vary over time without needing to consider new or deleted properties.
  - Change a column in a CSV file stored in a key-value store, and you must update all of your application's code! This isn't the case with a JSON document, where older code simply ignores new properties.

- **Complex structure handling:**
  - Rather than store eight items separately that will require eight reads, denormalize the data — write the data to the same record at ingestion time — so that only one read is needed later.
  - This does mean some data will be stored multiple times. An example is storing customer name in an order document.
  - Denormalization consumes more disk space than relational databases' normal form, but greatly increases query throughput. It's the NoSQL equivalent of a materialized view in a relational database.
  - However, with the current low cost of storage and the increasing demands of modern applications, it's much better to sacrifice storage for speed in reading data.

Managing User Information

- Much of the supporting data helps users access a system, tailor a service to their needs, or find other available services or products.
- **Delivering web advertisements:**
  - Although advertisements are critical to companies marketing their wares or services on the web, they aren't essential to many users' web-browsing experiences.
  - However, the loading time of web pages is important to them, and as soon as a slowly delivered ad starts adding to a page's load time, users start moving to alternative, faster, websites.
  - Serving advertisements fast is, therefore, a key concern.
  - Which advertisement is shown to which user depends on a very large number of factors, often determined by such factors as the user's tracked activity online, language, and location.
  - Companies that target their advertisements to the right customers receive more click-throughs, and thus more profit. However, the business of targeted advertising is increasingly scientific.
  - Key-value stores are used mainly by web advertisement companies.

- **Handling user sessions:**
  - A user session may track how a user walks through an application, adding data on each page.
  - The data can then be saved at the end of this journey in a single hit to the database, rather than in a sequence of small requests across many page requests.
  - Users often don’t mind waiting a couple of seconds after clicking a save button. Providing an effective user session on a website that has low latency has a couple of benefits:
    - ✓The user (soon to be customer!) receives good service.
    - ✓Partially complete data doesn’t get saved to your main back-end transactional database.
  - Websites use a cookie to track the user’s interaction with a website. A cookie is a small file linked to a unique ID, just like a record in a key-value store.
  - The server uses these cookies to identify that it already knows a user on their econd or subsequent requests, so the server needs to fetch a session using this data quickly.
  - In this way, when users log in, the websites recognize who they are, which pages they visit, and what information they’re looking for.
  - Key-value stores are, therefore, ideal for storing and retrieving session data at high speeds. The ability to tombstone (that is delete) data once a timestamp is exceeded is also useful.

- **Supporting personalization:**
  - User service personalization is where the front-end application is configured by users for their specific needs.
  - Using a key-value store with a composite key containing user id (not session id) and the service name allows you to store the personalization settings as a value, which makes lookups very quick and prevents the performance of your primary systems from being negatively affected.
- **High-Speed Data Caching:**
  - High-speed in-memory caching provides this caching capability without the need for a separate application level caching layer.
  - This reduces total cost of ownership and makes developing well-performing applications quicker and easier

High-Speed Key Access

- Key-value stores are all about speed. You can use various techniques to maximize that speed, from caching data, to having multiple copies of data, or using the most appropriate storage structures.
- **Caching data in memory:**
  - Because data is easily accessed when it's stored in random access memory (RAM), choosing a key-value store that caches data in RAM can significantly speed up your access to data, albeit at the price of higher server costs.
- **Replicating data to slaves:**
  - In key-value stores, a particular key is stored on one of the servers in the cluster. This process is called key partitioning.
  - This means that, if this key is constantly requested, this node will receive the bulk of requests.
  - This node, therefore, will be slower than your average request speed, potentially affecting the quality of service to your users.
  - To avoid this situation, some key-value stores support adding read-only replicas, also referred to as slaves. Redis, Riak, and Aerospike are good examples.
  - Replication allows the key to be stored multiple times across several servers, which increases response speed but at the cost of more hardware

- **Data modeling in key-value stores:**
  - Many key-value stores support only basic structures for their value types, leaving the application programmer with the job of interpreting the data.
  - Simple data type support typically includes strings, integers, JSON, and binary values.
  - For many use cases, this works well, but sometimes a slightly more granular access to data is useful. Redis, for example, supports the following data value types:
    1. String
    2. List
    3. Set
    4. Sorted set
    5. Hash maps
    6. Bit arrays
    7. Hyperlog logs
  - Sorted sets can be queried for matching ranges of values — much like querying an index of values sorted by date, which is very useful for searching for a subset of typed data.

- **Operating on data:**
  - Redis includes operations to increment and decrement key values directly, without having to do a read-modify-update (RMU) set of steps.
  - You can do so within a single transaction to ensure that no other application changes the value during an update.
  - These data-type specific operations include adding and removing items to lists and sets, too.
  - You can even provide autocomplete functionality on an application's user interface by using the Redis ZRANGEBYLEX command. This command retrieves a set of keys which partially matches a string.
- **Evaluating Redis:**
  - Redis prides itself on being a very lightweight but blazingly fast key-value store. It was originally designed to be an in-memory key-value store, but now boasts disk-based data storage.
  - You can use Redis to safeguard data by enabling AOF (append only file) mode and instructing Redis to force data to disk on each query (known as forced fsync flushing).
  - AOF does slow down writes, of course, but it provides a higher level of durability for data. Be aware, though, that it's still possible to lose up to one second of commands.
  - Also, Redis only recently added support for clustering.

Taking Advantage of Flash

- When you need incredibly fast writes, flash storage is called.
- Writing to RAM will get you, well, about as far as the size of your RAM. So having a very high-speed storage option immediately behind your server's RAM is a good idea.
- **Spending money for speed:**
  - Flash is expensive — more so than traditional spinning disk and RAM. It's possible to make do without flash by using RAID 10 spinning disk arrays, but these will get you only so far.
  - A logical approach is to look at how fast data streams into your database.
  - Perhaps provisioning 100 percent of the size of your store data for a spinning disk, 10 percent for flash, and one percent for RAM.
- **Context computing:**
  - Aerospike espouses a concept called context-aware computing. Context-aware computing is where you have a very short window of time to respond to a request, and the correct response is dictated by some properties of the user, such as age or products purchased. These properties could include:
  - Identity: Session IDs, cookies, IP addresses
  - Attributes: Demographic or geographic
  - Behavior: Presence (swipe, search, share), channels (web, phone),services (frequency, sophistication)

- Identity: Session IDs, cookies, IP addresses
- Attributes: Demographic or geographic
- Behavior: Presence (swipe, search, share), channels (web, phone),services (frequency, sophistication)
- Segments: Attitudes, values, lifestyle, history
- Transactions: Payments, campaigns

- **Evaluating Aerospike:**
  - Aerospike is the king of flash support. Rather than use the operating system’s file system support on top of flash, Aerospike natively accesses the flash.
  - This behavior provides Aerospike with maximum throughput, because it doesn’t have to wait for operating system function calls to be completed; it simply accesses the raw flash blocks directly.
  - Moreover, Aerospike can take advantage of the physical attributes of flash storage in order to eke out every last bit of performance.
  - It has enterprise-level features lacking in other databases, including the following:

- **Full ACID consistency**: Ensures data is safe and consistent.
- **Shared-nothing cluster**: Has synchronous replication to keep data consistent.
- **Automatic rebalancing**: Automatically moves some data to new nodes, evening out read times and allowing for scale out and scale back in a cluster.
- **Support for UDFs and Hadoop**: User defined functions can run next to the data for aggregation queries, and Hadoop Map/Reduce is supported for more complex requirements.
- **Secondary indexes**: Adds indexes on data value fields for fast querying.
- **Large data types**: Supports custom and large data types; allows for complex data models and use cases.
- **Automatic storage tier flushing on writes**: Flushes RAM to flash storage (SSDs) and disk when space on the faster tier is nearly exhausted.

Using Pluggable Storage

- There are times when you want to provide key-value style high speed access to data held in a relational database. This database could be, for example, Berkeley DB
- Providing key-value like access to data requires a key-value store to be layered directly over one of these other databases.
- Basically, you use another database as the storage layer, rather than a combination of a file system for storage and an ingestion pipeline for copying data from a relational database.
- This process simplifies providing a high speed key-value store while using a traditional relational database for storage.
- **Changing storage engines:**
  - Different workloads require different storage engines and performance characteristics. Aerospike is great for high ingest; Redis is great for high numbers of reads. Each is built around a specific use case.
  - Voldemort takes a different approach. Rather than treating the key-value store as a separate tier of data management, Voldemort treats the key-value store as an API and adds an in-memory caching layer, which means that you can plug into the back end that makes the most sense for your particular needs.
  - This capability combined with custom data types allows you to use a key-value store’s simple store/retrieve API to effectively pull back and directly cache information in a different back-end store.

- **Caching data in memory:**
  - Voldemort has a built-in in-memory cache, which decreases the load on the storage engine and increases query performance.
  - No need to use a separate caching layer such as Redis or Oracle's Coherence Java application data caching product on top.
  - The capability to provide high-speed storage tiering with caching is why LinkedIn uses Voldemort for certain high-performance use cases.
  - With Voldemort, you get the best of both worlds — a storage engine for your exact data requirements and a high-speed in-memory cache to reduce the load on that engine.
  - You also get simple key-value store store/retrieve semantics on top of your storage engine.
- **Evaluating Voldemort:**
  - Voldemort is built in the Java programming language, which requires a Java Native Interface (JNI) connector to be built for integration to most C or C++ based databases.
  - Voldemort has good integration with serialization frameworks, though.
  - Supported frameworks include Java serialization, Avro, Thrift, and Protocol Buffers.

- This means that the provided API wrappers match the familiar serialization method of each programming language, making the development of applications intuitive.
- Voldemort doesn’t handle consistency as well as other systems do. Voldemort uses the read repair approach, where inconsistent version numbers for the same record are fixed at read time, rather than being kept consistent at write time.
- There is also no secondary indexing or query support; Voldemort expects you to use the facilities of the underlying storage engine to cope with that use case.
- Also, Voldemort doesn’t have native database triggers or an alerting or event processing framework with which to build one.
- If you do need a key-value store that is highly available, is partition-tolerant, runs in Java, and uses different storage back ends, then Voldemort may be for you.

Separating Data Storage and Distribution

- Oracle has a data-caching approach in Coherence. It also inherited the Berkeley DB code. Oracle chose to use Berkeley DB to produce a distributed key-value NoSQL database.
- **Using Berkeley DB for single node storage:**
  - Berkeley DB, as the name suggests, is an open-source project that started at the University of California, Berkeley, between 1986 and 1994. It was maintained by Sleepycat Software, which was later acquired by Oracle.
  - The idea behind Berkeley DB was to create a hash table store with the best performance possible.
  - Berkeley DB stores a set of keys, where each key points to a value stored on disk that can be read and updated using a simple key-value API.
  - Berkeley DB was originally used by the Netscape browser but can now be found in a variety of embedded systems.
  - Now you can use it for almost every coding platform and language. An SQL query layer is available for Berkeley DB, too, opening it up to yet another use case. Berkeley DB comes in three versions:
    - The Berkeley DB version written in C is the one that's usually embedded in UNIX systems.
    - The Java Edition is also commonly embedded, including in the Voldemort key-value store.
    - A C++ edition is available to handle the storage of XML data.
  - Berkeley DB typically acts as a single-node database.

- **Distributing data:**
  - Oracle built a set of data distribution and high-availability code using NoSQL design ideas on top of Berkeley DB.
  - This approach makes Oracle NoSQL a highly distributed key-value store that uses many copies of the tried-and-true Berkeley DB code as the primary storage system.
  - Oracle NoSQL is most commonly used alongside the Oracle relational database management systems (RDBMS) and Oracle Coherence.
  - Oracle Coherence is a mid-tier caching layer, which means it lives in the application server with application business code.
  - Applications can offload the storage of data to Coherence, which in turn distributes the data across the applications' server clusters. Coherence works purely as a cache.
  - Oracle Coherence is commonly used to store data that may have been originally from an Oracle RDBMS, to decrease the operational load on the RDBMS.
  - Using Oracle NoSQL with Coherence or directly in your application mid-tier, you can achieve a similar caching capability.

- **Evaluating Oracle NoSQL:**
  - Despite claims that Oracle NoSQL is an ACID database product, by default, it’s an eventually consistent — non-ACID — database.
  - This means data read from read replica nodes can potentially be stale.
  - The client driver can alleviate this situation by requesting only the absolute latest data, which not surprisingly is called absolute consistency mode.
  - This setting reads only data from the master node for a particular key.
  - It’s also worth noting that in the default mode (eventually consistent), because of the lack of a consistency guarantee, application developers must perform a check-and-set (CAS) or read-modify-update (RMU) set of steps to ensure that an update to data is applied properly.
  - In addition, unlike Oracle’s RDBMS product, Oracle NoSQL doesn’t have a write Journal.
  - Oracle is plugging Oracle NoSQL into its other products. Oracle NoSQL provides a highly scalable layer for Oracle Coherence.
  - Many financial services firms, though, are looking at other NoSQL options to replace Coherence.
  - Another product that may be useful in the future is RDF Graph for Oracle NoSQL.
  - The concept of major and minor keys provide more of a two-layer tree model than a single layer key-value model.

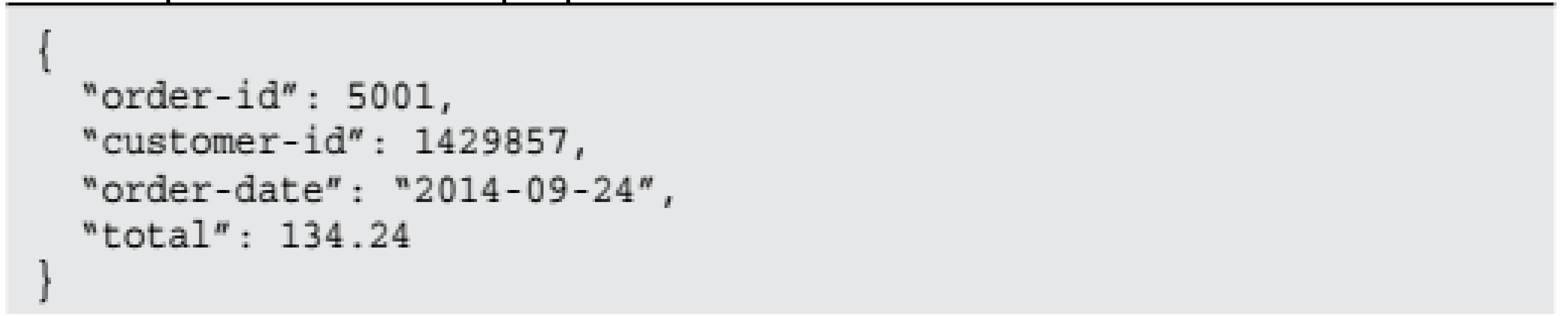
Handling Partitions

- A data partition is a mechanism for ensuring that data is evenly distributed across a cluster. On the other hand, a network partition occurs when two parts of the same database cluster cannot communicate.
- **Tolerating partitions:**
  - You have two choices when a network partition happens:

    ✓Continue, at some level, to service read and write operations.

    ✓"Vote off" one part of the partition and decide to fix the data later when both parts can communicate. This usually involves the cluster voting a read replica as the new master for each missing master partition node.
  - Riak handles writes when the primary partition server goes down by using a system called hinted handoff.
  - When data is originally replicated, the first node for a particular key partition is written to, along with (by default) two of the following neighbor nodes.
  - Riak employs yet another system called active anti-entropy to alleviate this problem. This system trawls through updated values and ensures that replicas are updated at some point, preferably sooner rather than later.
  - This helps to avoid conflicts on read while maintaining a high ingestion speed, which avoids a two-phase commit used by other NoSQL databases with master-slave, shared-nothing clustering support.
  - If a conflict on read does happen, Riak uses read repair to attempt to return only the latest data. Eventually though, and depending on the consistency and availability settings you use, the client application may be presented with multiple versions and asked to decide for itself.

- **Secondary indexing:**
  - Secondary indexes are indexes on specific data within a value.
  - Most key-value stores leave this indexing up to the application. However, Riak is different, employing a scheme called document-based partitioning that allows for secondary indexing.
  - Document-based partitioning assumes that you're writing JSON structures to the Riak database. You can then set up indexes on particular named properties within this JSON structure.

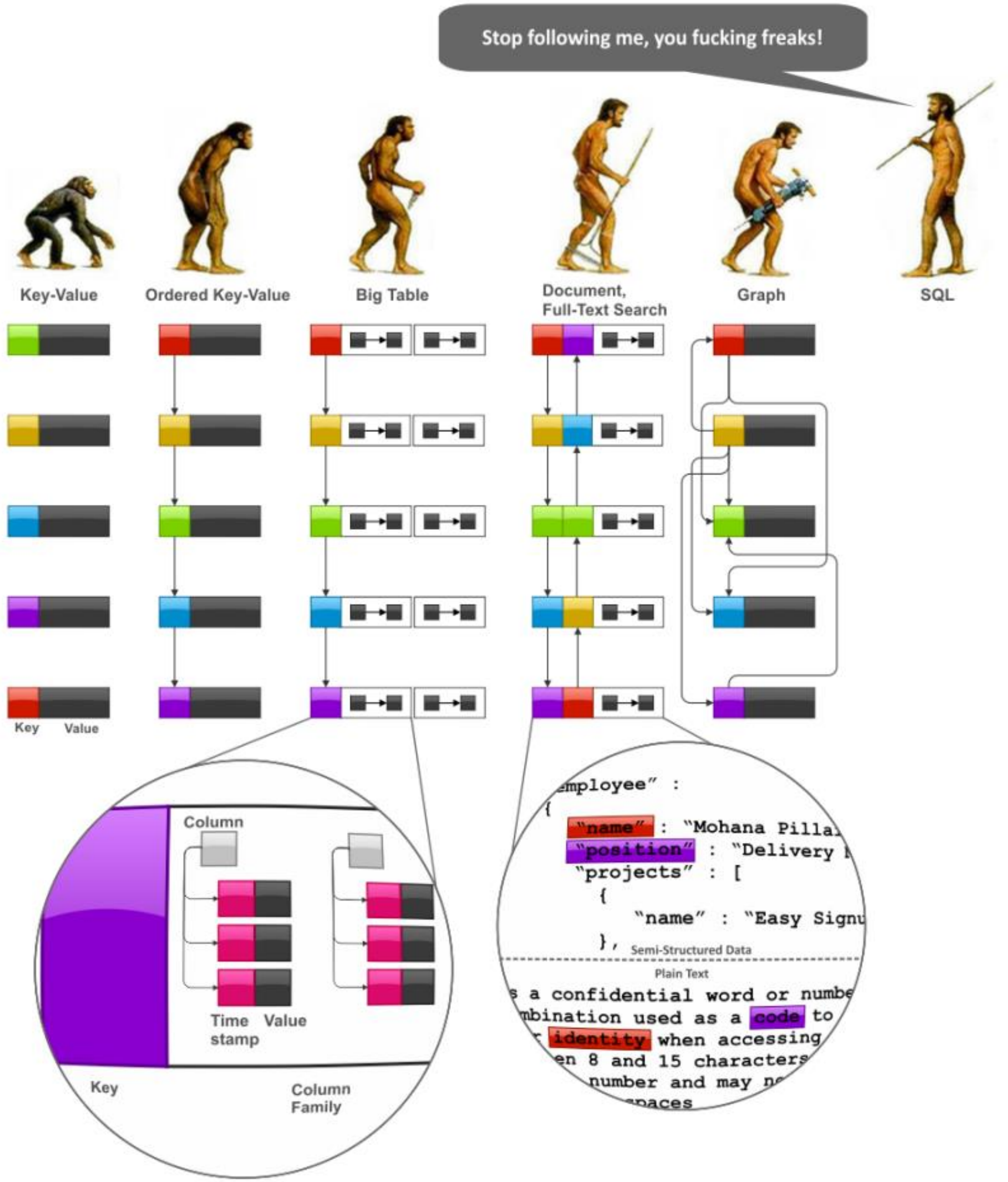


```
{
  "order-id": 5001,
  "customer-id": 1429857,
  "order-date": "2014-09-24",
  "total": 134.24
}
```

  - In most key-value stores, you create another bucket whose key is the combined customer number and month and the value is a list of order ids.
  - However, in Riak, you simply add a secondary index on both customer-id (integer) and order-date (date), which does take up extra storage space but has the advantage of being transparent to the application developer.
  - These indexes are also updated live — meaning there's no lag between updating a document value in Riak and the indexes being up to date.

- **Evaluating Riak:**
  - Riak is not an ACID-compliant database. Its configuration cannot be altered such that it runs in ACID compliance mode.
  - Clients can get inconsistent data during normal operations or during network partitions.
  - Riak trades absolute consistency for increased availability and partition tolerance.
  - Running Riak in strong consistency mode means that its read replicas are updated at the same time as the primary master.
  - This involves a two-phase commit — basically, the master node writing to the other nodes before it confirms that the write is complete.
  - At the time of this writing, Riak's strong consistency mode doesn't support secondary indexes or complex data types
  - Riak also uses a separate *sentinel process* to determine which node becomes a master in failover conditions.
  - Riak does have some nice features for application developers, such as secondary indexing and built-in JSON value support.
  - The Riak Control cluster monitoring tool also isn't highly regarded because of its lag time when monitoring clusters.

Module No.4
Key-Value Based Databases



Stop following me, you fucking freaks!
Key-Value
Ordered Key-Value
Big Table
Document, Full-Text Search
Graph
SQL
Key
Value
Column
Time stamp
Value
Key
Column Family
employee" :
{
"name" : "Mohana Pilla
"position" : "Delivery
"projects" : [
{
"name" : "Easy Signu
},
Semi-Structured Data
Plain Text
s a confidential word or numbe
mbination used as a code to
identity when accessing
en 8 and 15 characters
number and may n
spaces