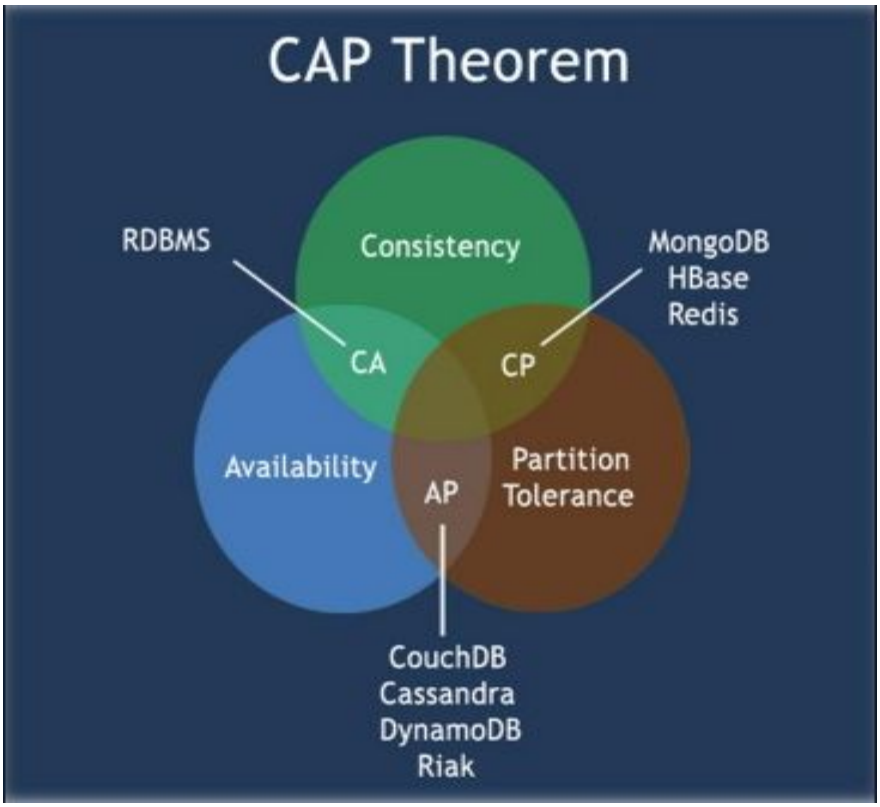
BDA SAMPLE sums

Big Data Analytics (University of Mumbai)

BDA UT2

1. Explain CAP theorem and how it is different from ACID properties

1. Consistency Availability Partition tolerance is three main aspects of the modern distributed data system.
2. The CAP theorem was coined by Eric Brewer in 2000 to address the standard way to maintain the network-based database systems.
3. In the era of petabyte-scale data, it became immensely important to develop and maintain distributed data systems to main the load.



The three aspects of CAP theorem are consistency, Availability and Partition tolerance. Let's first discuss all of these separately then we will join the pieces.

1. Consistency : According to this theorem, all connected nodes of the distributed system see the same value at the same times and partial transactions will not be saved. Suppose there are multiple steps inside a transaction and due to some malfunction some middle operation got corrupted, now if part of the connected nodes read the corrupted value, the data will be inconsistent and misleading. So according to the CAP principle, we will not allow such a transaction. A transaction cannot be executed partially. It will always be 'All or none'. If something goes wrong in between the execution of a transaction, the whole transaction needs to be rolled back.

2. Availability : According to this, the connected or distributed systems should remain operational all the time. There should be a response for every client request in the system irrespective of whether a particular node is being available or not. Though in practical scenarios it is purely based on the traffic requirements. The key point of this is every functioning node must return a response for all read and write requests in a reasonable amount of time.

3. Partition tolerance : According to the partition tolerance policy, if a subpart of the network is compromised, the entire distributed system should not go down. A system that is partition tolerance should recover fast from partial outrage. In practical scenarios partition tolerance cannot be an optional criterion, it should be maintained thoroughly. So adhering to the CAP theorem became always a choice between high consistency and high availability.

We cannot maintain all three principles of the CAP theorem simultaneously. Theoretically, we can maintain only CA, CP or AP.

- Consistency and Availability: This is systems with high consistency and very lesser downtime but the option of partition tolerance is not enforced.

--**For example**, network issues can down the entire distributed RDBMS system.

- Consistency and Partition tolerance: These systems adhere to high consistency and partition tolerance but there is a risk of some data being unavailable.

-- **Ex.** MongoDB.

- Availability and Partition tolerance: These systems adhere to high availability and partition tolerance but there is a risk of reading inconsistent data.

--**Ex**. Cassandra.

ACID

Atomic: All operations in a transaction succeed or every operation is rolled back.

Consistent: On the completion of a transaction, the database is structurally sound.

Isolated: Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

Durable: The results of applying a transaction are permanent, even in the presence of failures.

The main contribution is to use relaxed ACID properties in the CAP optimization process. This may be viewed as a bridge between the CAP theorem and the traditional ACID theory. Traditional ACID properties are weakened, but not completely dropped, in order to optimize CAP properties.

From a user point of view, systems should thus function as if both the traditional ACID properties and all the CAP properties were implemented.

ACID addresses an individual node's data consistency

CAP addresses cluster-wide data consistency

2. When it comes to big data how NoSQL scores over RDBMS.

1. Less need for ETL(Extract-Transform-Load)

NoSQL databases support storing data “as is.” Key value stores give you the ability to store simple data structures, whereas document NoSQL databases provide you with the ability to handle a range of flat or nested structures.

Most of the data flying between systems does so as a message. Typically, the data takes one of these formats:

- A binary object to be passed through a set of layers
- An XML document
- A JSON document

Being able to handle these formats natively in a range of NoSQL databases lessens the amount of code you have to convert from the source data format to the format that needs storing. This is called *extract, transform, and load* (ETL).

Using this approach, you greatly reduce the amount of code required to start using a NoSQL database. Moreover, because you don’t have to pay for updates to this “plumbing” code, ongoing maintenance costs are significantly decreased.

2. Support for unstructured text

The vast majority of data in enterprise systems is unstructured. Many NoSQL databases can handle indexing of unstructured text either as a native feature (MarkLogic Server) or an integrated set of services including Solr or Elasticsearch.

Being able to manage unstructured text greatly increases information and can help organizations make better decisions. For example, advanced uses include support for multiple languages with faceted search, snippet functionality, and word-stemming support. Advanced features also include support for dictionaries and thesauri.

Furthermore, using search alert actions on data ingest, you can extract named entities from directories such as those listing people, places, and organizations, which allows text data to be better categorized, tagged, and searched.

Entity enrichment services such as SmartLogic, OpenCalais, NetOwl, and TEMIS Luxid that combine extracted information with other information provide a rich interleaved information web and enhance efficient analysis and use.

3. Ability to handle change over time

Because of the schema agnostic nature of NoSQL databases, they’re very capable of managing change — you don’t have to rewrite ETL routines if the XML message structure between systems changes.

Some NoSQL databases take this a step further and provide a universal index for the structure, values, and text found in information. Microsoft DocumentDB and MarkLogic Server both provide this capability.

If a document structure changes, these indexes allow organizations to use the information immediately, rather than having to wait for several months before you can test and rewrite systems.

4. No reliance on SQL magic

Structured Query Language (SQL) is the predominant language used to query relational database management systems. Being able to structure queries so that they perform well has over the years become a thorny art. Complex multi table joins are not easy to write from memory.

Although several NoSQL databases support SQL access, they do so for compatibility with existing applications such as business intelligence (BI) tools. NoSQL databases support their own access languages that can interpret the data being stored, rather than require a relational model within the underlying database.

This more developer-centric mentality to the design of databases and their access application programming interfaces (API) are the reason NoSQL databases have become very popular among application developers.

Application developers don't need to know the inner workings and vagaries of databases before using them. NoSQL databases empower developers to work on what is required in the applications instead of trying to force relational databases to do what is required.

5. Ability to scale horizontally on commodity hardware

NoSQL databases handle partitioning (*sharding*) of a database across several servers. So, if your data storage requirements grow too much, you can continue to add inexpensive servers and connect them to your database cluster (*horizontal scaling*) making them work as a single data service.

Contrast this to the relational database world where you need to buy new, more powerful and thus more expensive hardware to scale up (*vertical scaling*). If you were to double the amount of data you store, you would easily quadruple the cost of the hardware you need.

Providing durability and high availability of a NoSQL database by using inexpensive hardware and storage is one of NoSQL's major assets. Being able to do so while providing generous scalability for many uses also doesn't hurt!

6. Breadth of functionality

Most relational databases support the same features but in a slightly different way, so they are all similar.

NoSQL databases, in contrast, come in four core types: key-value, columnar, document, and triple stores. Within these types, you can find a database to suit your particular (and peculiar!) needs. With so much choice, you're bound to find a NoSQL database that will solve your application woes.

7. Support for multiple data structures

Many applications need simple object storage, whereas others require highly complex and interrelated structure storage. NoSQL databases provide support for a range of data structures.

- Simple binary values, lists, maps, and strings can be handled at high speed in key-value stores.
- Related information values can be grouped in column families within Bigtable clones.
- Highly complex parent-child hierarchical structures can be managed within document databases.
- A web of interrelated information can be described flexibly and related in triple and graph stores.

8. Vendor choice

The NoSQL industry is awash with databases, though many have been around for less than ten years. For example, IBM, Microsoft, and Oracle only recently dipped their toes into this market. Consequently, many vendors are targeting particular audiences with their own brew of innovation.

Open-]source variants are available for most NoSQL databases, which enables companies to explore and start using NoSQL databases at minimal risk. These companies can then take their new methods to a production platform by using enterprise offerings.

9. No legacy code

Because they are so new, NoSQL databases don't have legacy code, which means they don't need to provide support for old hardware platforms or keep strange and infrequently used functionality updated.

NoSQL databases enjoy a quick pace in terms of development and maturation. New features are released all the time, and new and existing features are updated frequently (so NoSQL vendors don't need to maintain a very large code base). In fact, new major releases occur annually rather than every three to five years.

10. Executing code next to the data

NoSQL databases were created in the era of Hadoop. Hadoop's highly distributed file system (HDFS) and batch-processing environment (Map/Reduce) signaled changes in the way data is stored, queried, and processed.

Queries and processing work now pass to several servers, which provides high levels of parallelization for both ingest and query workloads. Being able to calculate aggregations next to the data has also become the norm.

You no longer need a separate data warehouse system that is updated overnight. With fast aggregations and query handling, analysis is passed to the database for execution next to the data, which means you don't have to ship a lot of data around a network to achieve locally combined analysis.

3. Explain the block diagram architecture of the Data stream Management System.

Traditional relational databases store and retrieve records of data that are static in nature. Further these databases do not perceive a notion of time unless time is added as an attribute to the database during designing the schema itself. While this model was adequate for most of the legacy applications and older repositories of information, many current and emerging applications require support for online analysis of rapidly arriving and changing data streams. This has prompted a deluge of research activity which attempts to build new models to manage streaming data. This has resulted in data stream management systems (DSMS), with an emphasis on continuous query languages and query evaluation.

Data Stream Model

A data stream is a real-time, continuous and ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is not possible to control the order in which the items arrive, nor it is feasible to locally store a stream in its entirety in any memory device. Further, a query over streams will actually run continuously over a period of time and incrementally return new results as new data arrives. Therefore, these are known as long-running, continuous, standing and persistent queries.

As a result of the above definition, we have the following characteristics that must be exhibited by any generic model that attempts to store and retrieve data streams.

1. The data model and query processor must allow both order-based and time-based operations (e.g., queries over a 10 min moving window or queries of the form which are the most frequently occurring data before a particular event and so on).

2. The inability to store a complete stream indicates that some approximate summary structures must be used. As a result, queries over the summaries may not return exact answers.

3. Streaming query plans must not use any operators that require the entire input before any results are produced. Such operators will block the query processor indefinitely

4. Any query that requires backtracking over a data stream is infeasible. This is due to the storage and performance constraints imposed by a data stream. Thus any online stream algorithm is restricted to make only one pass over the data.

5. Applications that monitor streams in real-time must react quickly to unusual data values. Thus, long-running queries must be prepared for changes in system conditions any time during their execution lifetime (e.g., they may encounter variable stream rates).

6. Scalability requirements dictate that parallel and shared execution of many

continuous queries must be possible.

An abstract architecture for a typical DSMS is depicted in Fig. 6.1. An input monitor may regulate the input rates, perhaps by dropping packets. Data are typically stored in three partitions:
1. Temporary working storage (e.g., for window queries).
2. Summary storage.
3. Static storage for meta-data (e.g., physical location of each source).

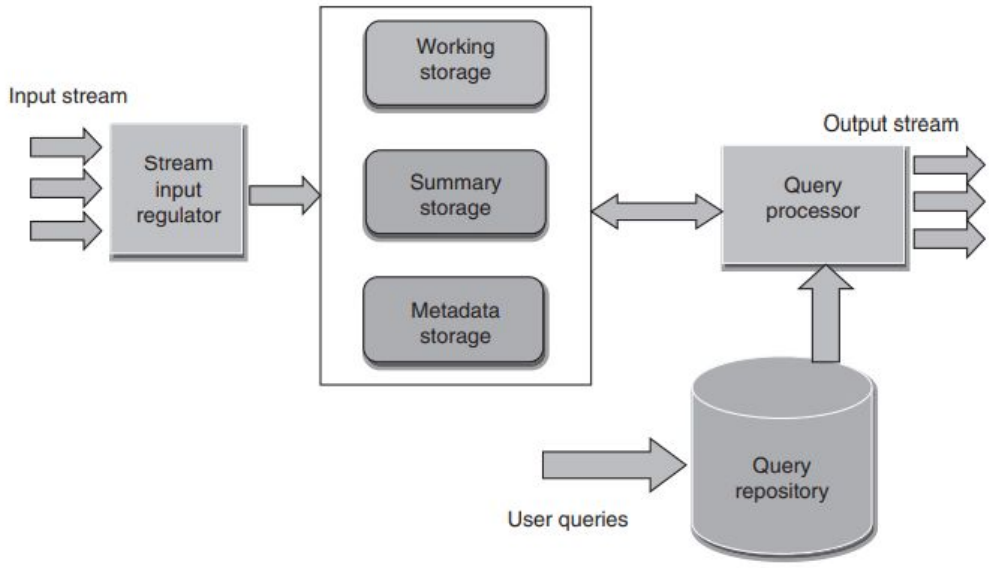


Figure 6.1 Abstract architecture for a typical DSMS.

Long-running queries are registered in the query repository and placed into groups for shared processing. It is also possible to pose one-time queries over the current state of the stream. The query processor communicates with the input monitor and may re-optimize the query plans in response to changing input rates. Results are streamed to the users or temporarily buffered.

4. Suppose a datastream consists of the integers:2 1 6 1 5 9 2 3 5,Let the function being used is

a)h(x)= 2x+3 mod 16

b)h(x)=4x+1 mod 16

c)h(x)=5x mod 16

Count Distinct Elements in a stream using FM algorithm.

Q. 2, 1, 6, 1, 5, 9, 2, 3, 5, Calculate distinct no. of elements in given data stream using FM algorithm

i) $h(x) = 2x+3 \bmod 16$
ii) $h(x) = 4x+1 \bmod 16$
iii) $5x \bmod 16$

i) $h(2) = 2(2)+3 \bmod 16 = 7$
$h(1) = 5$
$h(6) = 15$
$h(1) = 5$
$h(5) = 13$
$h(9) = 5$
$h(2) = 7$
$h(3) = 9$
$h(5) = 13$

Step 2:

$h(2) = 0111$
$h(1) = 0101$
$h(6) = 1111$
$h(1) = 0101$
$h(5) = 1101$
$h(9) = 0101$
$h(2) = 0111$
$h(3) = 1001$
$h(5) = 1101$

Step 3:

$h(2) = 0$
$h(1) = 0$
$h(6) = 0$
$h(1) = 0$
$h(5) = 0$
$h(9) = 0$
$h(2) = 0$
$h(3) = 0$
$h(5) = 0$

Step 4:

$r(a) = 0$

$\Rightarrow R = 2^r$

$= 2^0 = 1$

∴ There are 1 distinct elements for $h(x) = 2x+3 \bmod 16$

ii) $h(x) = 4x+1 \bmod 16$

$h(2) = 4(2)+1 \bmod 16 = 9$
$h(1) = 5$
$h(6) = 9$
$h(1) = 5$
$h(5) = 5$

$h(9) = 5$
$h(2) = 9$
$h(3) = 13$
$h(5) = 5$

Step 2:

$h(2) = 1001$
$h(1) = 0101$
$h(6) = 1001$
$h(1) = 0101$
$h(5) = 0101$

$h(9) = 0101$
$h(2) = 1001$
$h(3) = 1101$
$h(5) = 0101$

Step 3:

$h(2) = 0$
$h(1) = 0$
$h(6) = 0$
$h(1) = 0$
$h(5) = 0$

$h(9) = 0$
$h(2) = 0$
$h(3) = 0$
$h(5) = 0$

Step 4:

$r(a) = 0$

$\Rightarrow R = 2^r \Rightarrow 2^0 = 1$

∴ There are/is 1 distinct elements for $h(x) = 4x+1 \bmod 16$.

iii) $5x \bmod 16$

$h(2) = 5(2) \bmod 16 = 10$
$h(1) = 5$
$h(6) = 14$
$h(1) = 5$
$h(5) = 9$
$h(9) = 13$
$h(2) = 10$
$h(3) = 15$
$h(5) = 9$

Step 2:

$h(2) = 1010$
$h(1) = 0101$
$h(6) = 1110$
$h(1) = 0101$
$h(5) = 1001$
$h(9) = 1101$
$h(2) = 1010$
$h(3) = 1111$
$h(5) = 1001$

Step 3:

$h(2) = 1$
$h(1) = 0$
$h(6) = 1$
$h(1) = 0$
$h(5) = 0$
$h(9) = 0$
$h(2) = 1$
$h(3) = 0$
$h(5) = 0$

Step 4:

$r(a) = 1$

$\Rightarrow R = 2^r \Rightarrow 2^1 = 2$

$\therefore$ There are 2 distinct elements for $h(x) = 5x \bmod 16$

5. Explain different ways by which big data problems are handled by NoSQL.

1. Moving queries to the data, not data to the queries: With the exception of large graph databases, most NoSQL systems use commodity processors that each hold a subset of the data on their local shared-nothing drives. When a client wants to send a general query to all nodes that hold data, it's more efficient to send the query to each node than it is to transfer large datasets to a central processor. This simple rule helps you understand how NoSQL databases can have dramatic performance advantages over systems that weren't designed to distribute queries to the data nodes. Keeping all the data within each data node in the form of logical documents means that only the query itself and the final result need to be moved over a network. This keeps your big data queries fast.

2. Using hash rings to evenly distribute data on a cluster: One of the most challenging problems with distributed databases is figuring out a consistent way of assigning a document to a processing node. Using a hash ring technique to evenly distribute big data loads over many servers with a randomly generated 40-character key is a good way to evenly distribute a network load.

Hash rings are common in big data solutions because they consistently determine how to assign a piece of data to a specific processor. Hash rings take the leading bits of a document's hash value and use this to determine which node the document should be assigned. This allows any node in a cluster to know what node the data lives on and how to adapt to new assignment methods as your data grows. Partitioning keys into ranges and assigning different key ranges to specific nodes is known as keyspace management.

3. Using replication to scale reads: Databases use replication to make backup copies of data in real time. Using replication allows you to horizontally scale read requests

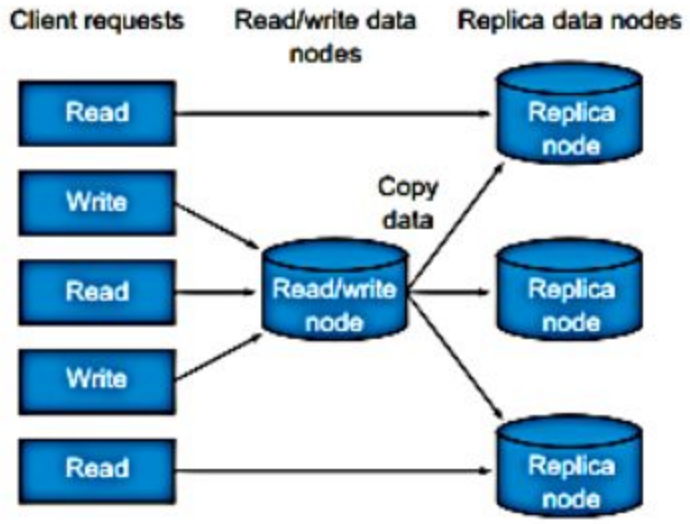


Figure: How you can replicate data to speed read performance in NoSQL systems.

This replication strategy works well in most cases. There are only a few times when you must be concerned about the lag time between a write to the read/write node and a client reading that same record from a replica. One of the most common operations after a write is a read of that same record. If a client does a write and then an immediate read from that same node, there's no problem. The problem occurs if a read occurs from a replica node

before the update happens. This is an example of an inconsistent read. The best way to avoid this type of problem is to only allow reads to the same write node after a write has been done. This logic can be added to a session or state management system at the application layer. Almost all distributed databases relax database consistency rules when a large number of nodes permit writes. If your application needs fast read/write consistency, you must deal with it at the application layer.

4. Letting the database distribute queries evenly to data nodes: In order to get high performance from queries that span multiple nodes, it's important to separate the concerns of query evaluation from query execution. Figure shows this structure:

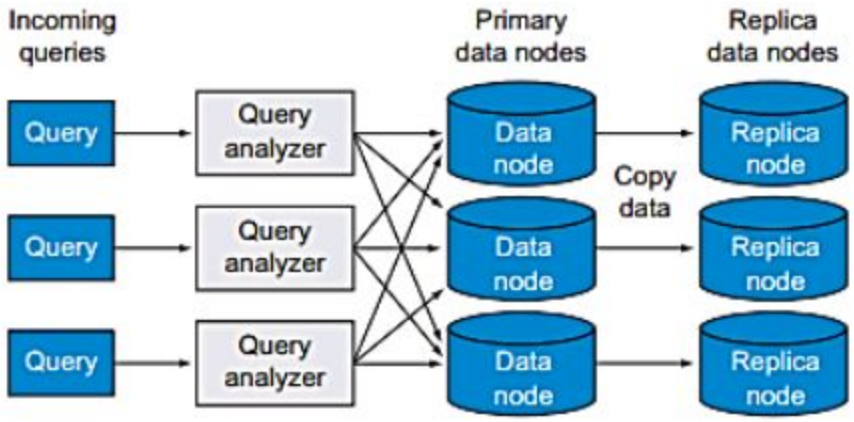


Figure NoSQL systems move the query to a data node, but don't move data to a query node. In this example, all incoming queries arrive at query analyzer nodes. These nodes then forward the queries to each data node. If they have matches, the documents are returned to the query node. The query won't return until all data nodes (or a response from a replica) have responded to the original query request. If the data node is down, a query can be redirected to a replica of the data node.

This approach is somewhat similar to the concept of federated search. Federated search takes a single query and distributes it to distinct servers and then combines the results together to give the user the impression they're searching a single system.

6. Discuss NoSQL architectural Pattern with examples

The key−value store, column family store, document store and graph store patterns can be modified based on different aspects of the system and its implementation. Database architecture could be distributed (manages single databases distributed in multiple servers located at various sites) or federated (manages independent and heterogeneous databases at multiple sites).

In the Internet of Things (IoT) architecture a virtual sensor has to integrate multiple data streams from real sensors into a single data stream. Results of the queries can be stored temporarily and consumed by the application or stored permanently if required for later use. For this, it uses data and sources-centric IoT middleware.
Scalable architectural patterns can be used to form new scalable architectures. For example, a combination of load balancer and shared-nothing architecture; distributed Hash Table and Content Addressable network (Cassandra); Publish/Subscribe (EventJava); etc.

The variations in architecture are based on system requirements like agility, availability (anywhere, anytime), intelligence, scalability, collaboration and low latency. Various technologies support the architectural strategies to satisfy the above requirement. For example, agility is given as a service using virtualization or cloud computing; availability is the service given by internet and mobility; intelligence is given by machine learning and predictive analytics; scalability (flexibility of using commodity machines) is given by Big Data Technologies/cloud platforms; collaboration is given by (enterprise-wide) social network application; and low latency (event driven) is provided by in memory databases.

7. Give two applications of counting the number of 1's in a given long stream of binary values. Discuss DGIM algorithm to count the number of 1s..

Applications of DGIM:

- In an e-commerce website there are thousands of products and millions of ransactions.
- We can have an vector of 0/1 to indicate if a product X is sold in any nth transaction.
- The query on this stream is like - how many times we have sold X in the last K sales.
- On the same line we can represent the users who purchased the products using 0/1
- So the query can be asked like how many users did purchased products in last K transactions.

Datar–Gionis–Indyk–Motwani presented an elegant and simple algorithm called DGIM, which uses O(log2N) bits to represent a window of N bits, and enables the estimation of the number of 1s in the window with an error of no more than 50%.
To begin we assume that new data elements are coming from the right and the elements at the left are ones already seen. We timestamp the active data elements from right to left, with the most recent element being at position 1.

Estimating the Number of 1s in a Window: We can estimate the number of 1s in a window of 0s and 1s by grouping the 1s into buckets. Each bucket has a number of 1s that is a power of 2; there are one or two buckets of each size, and sizes never decrease as we go back in time. By recording only the position and size of the buckets, we can represent the contents of a window of size N with O(log2N) space. We use DGIM algorithm to estimate the number of 1sw. This estimate can never be off by more than 50% of the true count of 1s.

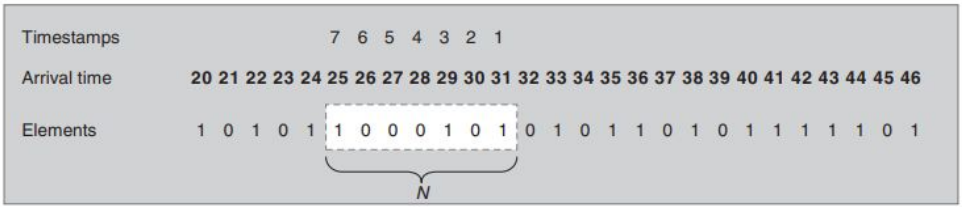


Figure 6.3 Snapshot of DGIM algorithm.

We divide the window into buckets, consisting of:

1. The timestamp of its right (most recent) 1.
2. The number of 1s in the bucket. This number must be a power of 2, and we refer to the number of 1s as the size of the bucket.

To begin, each bit of the stream has a timestamp, the position in which it arrives. The first bit has timestamp 1, the second has timestamp 2, and so on.

Since we only need to distinguish positions within the window of length N, we shall represent timestamps modulo N, so they can be represented by log2 N bits. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo N, then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is.

We divide the window into buckets, 5 consisting of:

1. The timestamp of its right (most recent) end.
2. The number of 1's in the bucket. This number must be a power of 2, and we refer to the number of 1's as the size of the bucket.

To represent a bucket, we need log2 N bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need log2 log2 N bits. The reason is that we know this number i is a power of 2, say 2j , so we can represent i by coding j in binary. Since j is at most log2 N, it requires log2 log2 N bits. Thus, O(logN) bits suffice to represent a bucket. There are six rules that must be followed when representing a stream by buckets.

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).

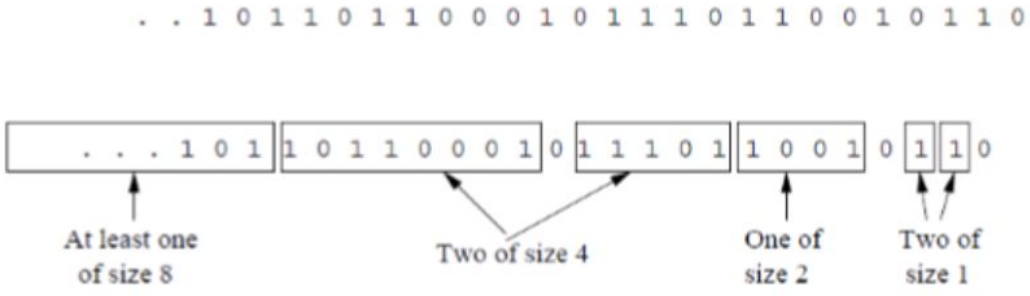


Figure: A bit-stream divided into buckets following the DGIM rules

8. Discuss Bloom's Filter for analysing data streams.

A Bloom filter is defined as a data structure designed to identify an element's presence in a set in a rapid and memory efficient manner.

A specific data structure named as a probabilistic data structure is implemented as a bloom filter. This data structure helps us to identify that an element is either present or absent in a set.

Bit Vector is implemented as a base data structure. Here's a small one we'll use to explain: Each empty cell in that table specifies a bit and the number below it is its index or position. To append an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the position or index of those hashes to 1.

Implementation of Bloom Filter:

Bloom filters support two actions, at first appending objects and keeping track of an object and next verifying whether an object has been seen before.

-- Appending objects to the Bloom filter

• We compute hash values for the object to append;

• We implement these hash-values to set certain bits in the Bloom filter state (hash value is the position of the bit
to set).

-- Verifying whether the Bloom filter contains an object −

• We compute hash values for the object to append;

• Next we verify whether the bits indexed by these hash values are set in the Bloom filter state.

- Consider: $|S| = m$, $|B| = n$
- Use k independent hash functions $h_1, \ldots, h_k$
- **Initialization:**
  - Set **B** to all **0s**
  - Hash each element $s \in S$ using each hash function h_i, set $B[h_i(s)] = 1$ (for each $i = 1, .., k$)

(**note:** we have a single array B!)

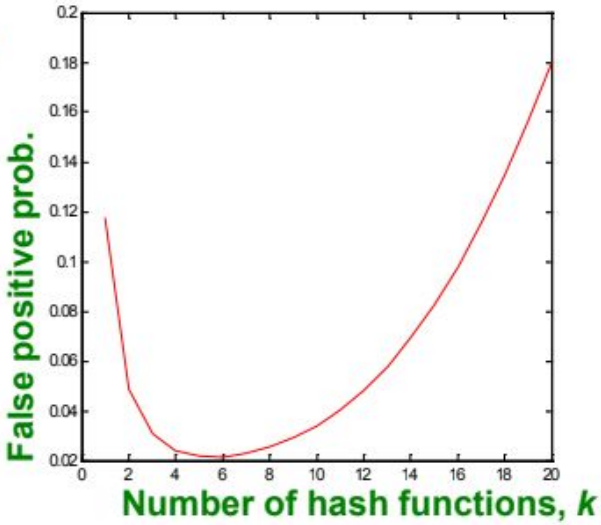
- **Run-time:**
  - When a stream element with key x arrives
    - If $B[h_i(x)] = 1$ <u>for all</u> $i = 1, \ldots, k$ then declare that x is in S
      - That is, x hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element x

Bloom Filter -- Analysis

- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of **1**s is $(1 - e^{-km/n})$
- But we have k independent hash functions and we only let the element x through **if all** k hash element x to a bucket of value **1**
- So, false **positive probability** $= (1 - e^{-km/n})^k$

Bloom Filter – Analysis (2)

- m = 1 billion, n = 8 billion
  - **k = 1**: $(1 - e^{-1/8}) = 0.1175$
  - **k = 2**: $(1 - e^{-1/4})^2 = 0.0493$
- **What happens as we keep increasing k?**



- "Optimal" value of k: n/m ln(2)
  - **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**
    - **Error at k = 6**: $(1 - e^{-1/6})^2 = 0.0235$

Bloom Filter: Wrap-up

- **Bloom filters guarantee no false negatives, and use limited memory**
  - Great for pre-processing before more expensive checks
- **Suitable for hardware implementation**
  - Hash function computations can be parallelized
- Is it better to have **1** big **B** or ***k*** small **B**s?
  - **It is the same:** $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping **1 big B** is simpler

9. Discuss NoSQL business drivers.

Enterprises today need highly reliable, scalable and available data storage across a configurable set of systems that act as storage nodes. The needs of organizations are changing rapidly. Many organizations operating with single CPU and Relational database management systems (RDBMS) were not able to cope up with the speed in which information needs to be extracted. Businesses have to capture and analyze a large amount of variable data, and make immediate changes in their business based on their findings.

Figure 3.1 shows RDBMS with the business drivers velocity, volume, variability and agility necessitates the emergence of NoSQL solutions. All of these drivers apply pressure to a single CPU relational model and eventually make the system less stable.

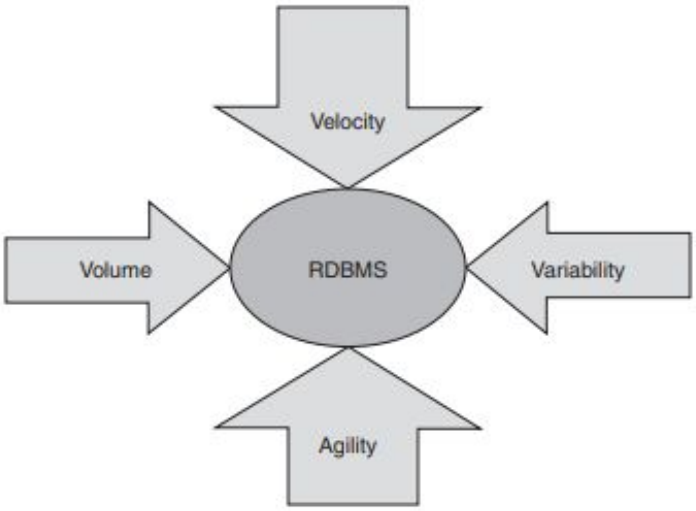


Figure 3.1 NoSQL business drivers.

Volume : There are two ways to look into data processing to improve performance. If the key factor is only speed, a faster processor could be used. If the processing involves complex (heavy) computation, Graphic Processing Unit (GPU) could be used along with the CPU. But the volume of data is limited to on-board GPU memory. The main reason for organizations to look at an alternative to their current RDBMSs is the need to query big data. The need for horizontal scaling made organizations move from serial to distributed parallel processing where big data is fragmented and processed using clusters of commodity machines. This is made possible by the development of technologies like Apache Hadoop, HDFS, MapR, HBase, etc

Velocity : Velocity becomes the key factor when frequency in which online queries to the database made by social networking and e-commerce web sites have to be read and written in real time. Many single CPU, RDBMS systems are unable to cope up with the demands of real-time inserts. RDBMS systems frequently index on many columns that decrease the system performance. For example, when online shopping sites introduce great discount schemes, the random bursts in web traffic will slow down the response for every user and tuning these systems as demand increases can be costly when both high read and write is required.

Variability : Organizations that need to capture and report on certain uncommon data, struggle when attempting to use RDBMS fixed schema. For example, if a business process wants to store a few special attributes for a few set of customers,then it needs to alter its schema definition. If a change is made to the schema, all customer rows within the database will also have this column. If there is no value related to this for most of the customers, then the row column representation will have a sparse matrix. In addition to this, new columns to an RDBMS require to execute ALTER TABLE command. This cannot be done on the fly since the present executing transaction has to complete and database has to be closed, and then schema can be altered. This process affects system availability, which means losing business.

Agility : The process of storing and retrieving data for complex queries in RDBMS is

quite cumbersome. If it is a nested query, data will have nested and repeated subgroups of data structures that are included in an object-relational mapping layer. This layer is responsible to generate the exact combination of SELECT, INSERT, DELETE and UPDATE SQL statements to move the object data from and to the backend RDBMS layer. This process is not simple and requires experienced developers with the knowledge of object-relational frameworks such as Java Hibernate. Even then, these change requests can cause slowdowns in implementation and testing.

Desirable features of NoSQL that drive business are listed below:

1. 24 × 7 Data availability: both function and data are to be replicated so that if database servers or "nodes" fail, the other nodes in the system are able to continue with operations without data loss. System updates can be made dynamically without having to take the database offline.

2. Location transparency: The ability to read and write to a storage node regardless of where that I/O operation physically occurs is termed as "Location Transparency or Location Independence".

3. Schema-less data model: NoSQL database system is a schema-free flexible data model that can easily accept all types of structured, semi-structured and unstructured data.

4. Modern day transaction analysis: Most of the transaction details relate to customer profile, reviews on products, branding, reputation, building business strategy, trading decisions, etc. that do not require ACID transactions. the "Consistency" is stated in the CAP theorem that signifies the immediate or eventual consistency of data across all nodes that participate in a distributed database.

5. Architecture that suits big data: NoSQL solutions provide modern architectures for applications that require high degrees of scale, data distribution and continuous availability.

10. Short Note on

a) Google's Big Table

Google's motivation for developing BigTable is driven by its need for massive scalability, better performance characteristics, and ability to run on commodity hardware. Each time when a new service or increase in load happens, its solution BigTable would result in only a small incremental cost. Volume of Google's data generally is in petabytes and is distributed over 100,000 nodes.

BigTable is built on top of Google's other services that have been in active use since 2005, namely, Google File System, Scheduler, MapReduce and Chubby Lock Service. BigTable is a column-family database which is designed to store data tables (sparse matrix of row, column values) as a section of column of data. It is distributed (high volume of data), persistent, multidimensional sorted map. The map is indexed by a row key, column key and a timestamp (int64).

Google Bigtable is a distributed, column-oriented data store created by Google Inc. to handle very large amounts of structured data associated with the company's Internet search and Web services operations.Bigtable was designed to support applications requiring massive scalability; from its first iteration, the technology was intended to be used with petabytes of data. The database was designed to be deployed on clustered systems and uses a simple data model that Google has described as "a sparse, distributed, persistent multidimensional sorted map." Data is assembled in order by row key, and indexing of the map is arranged according to row, column keys and timestamps. Compression algorithms help achieve high capacity. Google Bigtable serves as the database for applications such as the Google App Engine Datastore, Google Personalized Search, Google Earth and Google Analytics. Google has maintained the software as a proprietary, in-house technology. Nevertheless, Bigtable has had a large impact on NoSQL database design. Google software developers publicly disclosed Bigtable details in a technical paper presented at the USENIX Symposium on Operating Systems and Design Implementation in 2006. Google's thorough description of Bigtable's inner workings has allowed other organizations and open source development teams to create Bigtable derivatives, including the Apache HBase database, which is built to run on top of the Hadoop Distributed File System (HDFS). Other examples include Cassandra, which originated at Facebook Inc., and Hypertable, an open source technology that is marketed in a commercial version as an alternative to HBase

b) Amazon DynamoDB

Amazon.com has one of the largest e-commerce operations in the world. Customers from all around the world shop all hours of the day. So the site has to be up 24 × 7. Initially Amazon used the RDBMS system for shopping cart and checkout systems. Amazon DynamoDB, a NoSQL store, brought a turning point.

DynamoDB addresses performance, scalability and reliability, the core problems of RDBMS when it comes to growing data. Developers can store unlimited amounts of data by creating a database table and DynamoDB automatically saves it at multiple servers specified by the customer and also replicates them across multiple “Availability” Zones. It can handle the data and traffic while maintaining consistent, fast performance. The cart data and session data are stored in the key−value store and the final (completed) order is saved in the RDBMS as shown in Fig. 3.3.

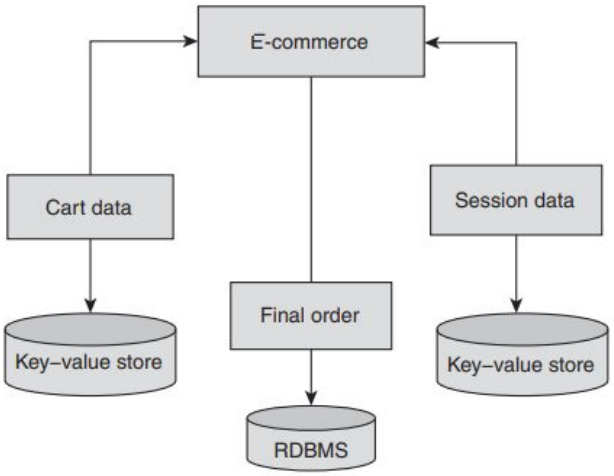


Figure 3.3 E-commerce shopping cart uses key–value store.

The salient features of key−value store are as follows:

1. Scalable: If application's requirements change, using the AWS Management Console or the DynamoDB APIs table throughput capacity can be updated.

2. Automated storage scaling: Using the DynamoDB write APIs, more storage can be obtained, whenever additional storage is required.

3. Distributed horizontally shared nothing: Seamlessly scales a single table over hundreds of commodity servers.

4. Built-in fault tolerance: DynamoDB automatically replicates data across multiple available zones in synchronous manner.

5. Flexible: DynamoDB has a schema-free format. Multiple data types (strings, numbers, binary and sets) can be used and each data item can have a different number of attributes.

6. Efficient indexing: Every item is identified by a primary key. It allows secondary indexes on non-key attributes, and query data using an alternate key.

7. Strong consistency, Atomic counters: DynamoDB ensures strong consistency on reads (only the latest values are read). DynamoDB service also supports atomic counters to atomically increment or decrement numerical attributes with a single API call.

8. Secure: DynamoDB uses cryptography to authenticate users. Access control for users within an organization can be secured by integrating with AWS Identity and Access Management.

9. Resource consumption monitoring: AWS Management Console displays request throughput and latency for each DynamoDB table, since it is integrated with CloudWatch.

10. Data warehouse − Business intelligence facility: Amazon Redshift Integration − data from DynamoDB tables can be loaded into Amazon Redshift (data warehouse service).

11. MapReduce integration: DynamoDB is also integrated with Amazon Elastic MapReduce to perform complex analytics (hosted pay-as-you-go Hadoop framework on AWS).