FullStack.Cafe - Kill Your Tech Interview

Q1: What do you understand by NoSQL databases? Explain. ☆

Topics: NoSQL

Answer:

At the present time, the internet is loaded with big data, big users, big complexity etc. and also becoming more complex day by day. NoSQL is answer of all these problems; It is not a traditional database management system, not even a relational database management system (RDBMS). NoSQL stands for “Not Only SQL”. NoSQL is a type of database that can handle and sort all type of unstructured, messy and complicated data. It is just a new way to think about the database.

Q2: What are NoSQL databases? What are the different types of NoSQL databases? ☆

Topics: NoSQL

Answer:

A NoSQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases (like SQL, Oracle, etc.).

Types of NoSQL databases:

- Document Oriented
- Key Value
- Graph
- Column Oriented

Q3: What are the advantages of NoSQL over traditional RDBMS? ☆☆

Topics: NoSQL Databases

Answer:

NoSQL is better than RDBMS because of the following reasons/properities of NoSQL:

- It supports semi-structured data and volatile data
- It does not have schema
- Read/Write throughput is very high
- Horizontal **scalability** can be achieved easily
- Will support Bigdata in volumes of Terra Bytes & Peta Bytes
- Provides good support for Analytic tools on top of Bigdata
- Can be hosted in cheaper hardware machines
- In-memory caching option is available to increase the performance of queries
- Faster development life cycles for developers

Still, **RDBMS is better** than NoSQL for the following reasons/properties of RDBMS:

- Transactions with **ACID** properties - Atomicity, Consistency, Isolation & Durability
- Adherence to **Strong Schema** of data being written/read
- Real time query management (in case of data size < 10 Tera bytes)
- Execution of complex queries involving **join** & **group by** clauses

Q4: Explain difference between scaling horizontally and vertically for databases ☆☆

Topics: NoSQL

Answer:

- Horizontal scaling means that you scale by adding more machines into your pool of resources whereas
- Vertical scaling means that you scale by adding more power (CPU, RAM) to an existing machine.

In a database world horizontal-scaling is often based on the partitioning of the data i.e. each node contains only part of the data, in vertical-scaling the data resides on a single node and scaling is done through multi-core i.e. spreading the load between the CPU and RAM resources of that machine.

Good examples of horizontal scaling are Cassandra, MongoDB, Google Cloud Spanner. and a good example of vertical scaling is MySQL - Amazon RDS (The cloud version of MySQL).

Q5: How does column-oriented NoSQL differ from document-oriented? ☆☆☆

Topics: NoSQL

Answer:

The main difference is that **document stores** (e.g. MongoDB and CouchDB) allow arbitrarily complex documents, i.e. subdocuments within subdocuments, lists with documents, etc. whereas **column stores** (e.g. Cassandra and HBase) only allow a fixed format, e.g. strict one-level or two-level dictionaries.

For example a document-oriented database (like MongoDB) inserts whole documents (typically JSON), whereas in Cassandra (column-oriented db) you can address individual columns or supercolumns, and update these individually, i.e. they work at a different level of granularity. Each column has its own separate timestamp/version (used to reconcile updates across the distributed cluster).

The Cassandra column values are just bytes, but can be typed as ASCII, UTF8 text, numbers, dates etc. You could use Cassandra as a primitive document store by inserting columns containing JSON - but you wouldn't get all the features of a real document-oriented store.

Q6: What does Document-oriented vs. Key-Value mean in context of NoSQL? ☆☆☆

Topics: NoSQL

Answer:

A **key-value** store provides the simplest possible data model and is exactly what the name suggests: it's a storage system that stores values indexed by a key. You're limited to query by key and the values are opaque, the store doesn't know anything about them. This allows very fast read and write operations (a simple disk

access) and I see this model as a kind of non volatile cache (i.e. well suited if you need fast accesses by key to long-lived data).

A **document-oriented** database extends the previous model and values are stored in a structured format (a document, hence the name) that the database can understand. For example, a document could be a blog post and the comments and the tags stored in a denormalized way. Since the data are transparent, the store can do more work (like indexing fields of the document) and you're not limited to query by key. As I hinted, such databases allows to fetch an entire page's data with a single query and are well suited for content oriented applications (which is why big sites like Facebook or Amazon like them).

Other kinds of NoSQL databases include column-oriented stores, graph databases and even object databases.

Q7: When should I use a NoSQL database instead of a relational database? ☆☆☆

Topics: NoSQL Databases

Answer:

Relational databases enforces ACID. So, you will have schema based transaction oriented data stores. It's proven and suitable for 99% of the real world applications. You can practically do anything with relational databases.

But, there are limitations on speed and scaling when it comes to massive high availability data stores. For example, Google and Amazon have terabytes of data stored in big data centers. Querying and inserting is not performant in these scenarios because of the blocking/schema/transaction nature of the RDBMs. That's the reason they have implemented their own databases (actually, **key-value stores**) for massive performance gain and scalability.

If you need a NoSQL db you usually know about it, possible reasons are:

- client wants 99.999% availability on a high traffic site.
- your data makes no sense in SQL, you find yourself doing multiple JOIN queries for accessing some piece of information.
- you are breaking the relational model, you have CLOBs that store denormalized data and you generate external indexes to search that data.

Q8: When would you use NoSQL? ☆☆☆

Topics: NoSQL Databases

Answer:

It depends from some general points:

- NoSQL is typically good for unstructured/"schemaless" data - usually, you don't need to explicitly define your schema up front and can just include new fields without any ceremony
- NoSQL typically favours a denormalised schema due to no support for JOINs per the RDBMS world. So you would usually have a flattened, denormalized representation of your data.
- Using NoSQL doesn't mean you could lose data. Different DBs have different strategies. e.g. MongoDB - you can essentially choose what level to trade off performance vs potential for data loss - best performance = greater scope for data loss.
- It's often very easy to scale out NoSQL solutions. Adding more nodes to replicate data to is one way to a) offer more scalability and b) offer more protection against data loss if one node goes down. But again, depends on the NoSQL DB/configuration. NoSQL does not necessarily mean "data loss" like you infer.

- IMHO, complex/dynamic queries/reporting are best served from an RDBMS. Often the query functionality for a NoSQL DB is limited.
- It doesn't have to be a 1 or the other choice. My experience has been using RDBMS in conjunction with NoSQL for certain use cases.
- NoSQL DBs often lack the ability to perform atomic operations across multiple "tables".

Q9: What Is BASE Property Of A System? ☆☆☆☆

Topics: Databases Software Architecture NoSQL

Answer:

BASE properties are the common properties of recently evolved NoSQL databases. According to CAP theorem, a BASE system does not guarantee consistency. This is a contrived acronym that is mapped to following property of a system in terms of the CAP theorem:

- **Basically available** indicates that the system is guaranteed to be available
- **Soft state**indicates that the state of the system may change over time, even without input. This is mainly due to the eventually consistent model.
- **Eventual consistency** indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

Q10: Explain use of transactions in NoSQL ☆☆☆☆

Topics: NoSQL

Answer:

NoSQL covers a diverse set of tools and services, including key-value-, document, graph and wide-column stores. They usually try improving scalability of the data store, usually by distributing data processing. Transactions require ACID properties of how DBs perform user operations. ACID restricts how scalability can be improved: most of the NoSQL tools relax consistency criteria of the operatioins to get fault-tolerance and availability for scaling, which makes implementing ACID transactions very hard.

A commonly cited theoretical reasoning of distributed data stores is the CAP theorem: consistency, availability and partition tolerance cannot be achieved at the same time.

A new, weaker set of requirements replacing ACID is BASE ("basically avalilable, soft state, eventual consistency"). However, eventually consistent tools ("eventually all accesses to an item will return the last updated value") are hardly acceptable in transactional applications like banking.

Generally speaking, NoSQL solutions have lighter weight transactional semantics than relational databases, but still have facilities for atomic operations at some level. Generally, the ones which do master-master replication provide less in the way of consistency, and more availability. So one should choose the right tool for the right problem.

Many offer transactions at the single document (or row etc.) level. For example with MongoDB there is atomicity at the single document - but documents can be fairly rich so this usually works.

Q11: How do you track record relations in NoSQL? ☆☆☆☆

Topics: NoSQL Databases

Answer:

All the answers for how to store many-to-many associations in the "NoSQL way" reduce to the same thing: **storing data redundantly**.

In NoSQL, you don't design your database based on the relationships between data entities. You design your database based on the queries you will run against it. Use the same criteria you would use to denormalize a relational database: if it's more important for data to have cohesion (think of values in a comma-separated list instead of a normalized table), then do it that way.

But this inevitably optimizes for one type of query (e.g. comments by any user for a given article) at the expense of other types of queries (comments for any article by a given user). If your application has the need for both types of queries to be equally optimized, you should not denormalize. And likewise, you should not use a NoSQL solution if you need to use the data in a relational way.

There is a risk with denormalization and redundancy that redundant sets of data will get out of sync with one another. This is called an anomaly. When you use a normalized relational database, the RDBMS can prevent anomalies. In a denormalized database or in NoSQL, it becomes your responsibility to write application code to prevent anomalies.

One might think that it'd be great for a NoSQL database to do the hard work of preventing anomalies for you. There is a paradigm that can do this - the relational paradigm.

Q12: Explain eventual consistency in context of NoSQL ☆☆☆☆

Topics: NoSQL Databases

Answer:

Think about **Eventual consistency** (as opposed to Strict Consistency/ACID compliance) as:

1. Your data is replicated on multiple servers
2. Your clients can access any of the servers to retrieve the data
3. Someone writes a piece of data to one of the servers, but it wasn't yet copied to the rest
4. A client accesses the server with the data, and gets the most up-to-date copy
5. A different client (or even the same client) accesses a different server (one which didn't get the new copy yet), and gets the old copy

Basically, because it takes time to replicate the data across multiple servers, requests to read the data might go to a server with a new copy, and then go to a server with an old copy. The term "eventual" means that eventually the data will be replicated to all the servers, and thus they will all have the up-to-date copy.

Eventual consistency is a must if you want low latency reads, since the responding server must return its own copy of the data, and doesn't have time to consult other servers and reach a mutual agreement on the content of the data.

The reason why so many NoSQL systems have **eventual consistency** is that virtually all of them are designed to be distributed, and with fully distributed systems there is super-linear overhead to maintaining strict consistency (meaning you can only scale so far before things start to slow down, and when they do you need to throw exponentially more hardware at the problem to keep scaling).

Q13: Explain how would you keep document change history in NoSQL DB? ☆☆☆☆

Topics: NoSQL

Answer:

There are some solution for that:

1. **Create a new version of the document on each change** - Add a version number to each document on change. The major drawback is that the entire document is duplicated on each change, which will result in a lot of duplicate content being stored when you're dealing with large documents. This approach is fine though when you're dealing with small-sized documents and/or don't update documents very often.
2. **Only store changes in a new version** - For that store only the changed fields in a new version. Then you can 'flatten' your history to reconstruct any version of the document. This is rather complex though, as you need to track changes in your model and store updates and deletes in a way that your application can reconstruct the up-to-date document. This might be tricky, as you're dealing with structured documents rather than flat SQL tables.
3. **Store changes within the document** - Each field can also have an individual history. Reconstructing documents to a given version is much easier this way. In your application you don't have to explicitly track changes, but just create a new version of the property when you change its value.

```
{
  _id: "4c6b9456f61f000000007ba6"
  title: [
    { version: 1, value: "Hello world" },
    { version: 6, value: "Foo" }
  ],
  body: [
    { version: 1, value: "Is this thing on?" },
    { version: 2, value: "What should I write?" },
    { version: 6, value: "This is the new body" }
  ],
  tags: [
    { version: 1, value: [ "test", "trivial" ] },
    { version: 6, value: [ "foo", "test" ] }
  ],
  comments: [
    {
      author: "joe", // Unversioned field
      body: [
        { version: 3, value: "Something cool" }
      ]
    },
    {
      author: "xxx",
      body: [
        { version: 4, value: "Spam" },
        { version: 5, deleted: true }
      ]
    },
    {
      author: "jim",
      body: [
        { version: 7, value: "Not bad" },
        { version: 8, value: "Not bad at all" }
      ]
    }
  ]
}
```

4. **Variation on Store changes within the document** - Instead of storing versions against each key pair, the current key pairs in the document always represents the most recent state and a 'log' of changes is stored within a history array. Only those keys which have changed since creation will have an entry in the log.

```
{
  _id: "4c6b9456f61f000000007ba6"
  title: "Bar",
  body: "Is this thing on?",
  tags: [ "test", "trivial" ],
  comments: [
    { key: 1, author: "joe", body: "Something cool" },
    { key: 2, author: "xxx", body: "Spam", deleted: true },
```

```
      { key: 3, author: "jim", body: "Not bad at all" }
    ],
    history: [
      {
        who: "joe",
        when: 20160101,
        what: { title: "Foo", body: "What should I write?" }
      },
      {
        who: "jim",
        when: 20160105,
        what: { tags: ["test", "test2"], comments: { key: 3, body: "Not baaad at all" }
      }
    ]
  }
```

Q14: Explain BASE terminology in a context of NoSQL ☆☆☆☆

Topics: NoSQL

Answer:

The **BASE** acronym is used to describe the properties of certain databases, usually NoSQL databases. It's often referred to as the opposite of ACID. The BASE acronym was defined by Eric Brewer, who is also known for formulating the CAP theorem.

The **CAP** theorem states that a distributed computer system cannot guarantee all of the following three properties at the same time:

- Consistency
- Availability
- Partition tolerance

A **BASE** system gives up on *consistency*.

- Basically *available* indicates that the system does guarantee availability, in terms of the CAP theorem.
- *Soft state* indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
- *Eventual consistency* indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

Q15: What is *Selectivity* of the query in MongoDB? ☆☆☆☆

Topics: MongoDB NoSQL

Answer:

Selectivity is the ability of a query to **narrow results using the index**. Effective indexes are more selective and allow MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

Suppose you have a field called `status` where the possible values are `new` and `processed`. If you add an index on `status` you've created a **low-selectivity** index. The index will be of little help in locating records.

A better strategy, depending on your queries, would be to create a **compound index** that includes the low-selectivity field and another field. For example, you could create a compound index on `status` and `created_at`.

Q16: Explain the differences in conceptual data design with NoSQL databases? ☆☆☆☆☆

Topics: NoSQL Databases

Problem:

What's easier, what's harder, what can't be done at all?

Solution:

I'm answering this with MongoDB in the back of my mind, but I would presume most would be true for other DBs also.

Harder:

- *Consistency* is not handled by the database but must be dealt with in the application. Less guarantees means easier migration, fail-over and better scalability at the cost of a more complicated application. An application has to deal with conflicts and inconsistencies.
- *Links* which cross documents (or key/value) have to be dealt with on application level also.
- SQL type of databases have IDEs which are much more mature. You get a lot of support libraries (although the layering of those libraries make things much more complex than needed for SQL).
- Keep related data together in the same document could be tricky, since there is nothing corresponding to a *join*.
- Map/reduce as a means of querying a database is unfamiliar, and requires a lot more thinking than writing SQL.

Easier:

- Faster if you know your *data access patterns* (views or specific queries).
- Migration / Fail-over is easier for the database since no promises are made to you as an application programmer. Although you get eventual consistency.
- One key / value is much easier to understand than one row from a table. All the (tree) relations are already in, and complete objects can be recognized.
- No designing DB tables
- No ODBC/JDBC intermediate layer, all queries and transactions over http
- Simple DB-to-object mapping from JSON, which is almost trivial compared to the same in SQL

Q17: Is the C in ACID is not the C in CAP? ☆☆☆☆☆

Topics: CAP Theorem Databases NoSQL

Answer:

For Database Systems - the Consistency in ACID properties is part of the acronym:

- A - Atomicity
- C - Consistency
- I - Isolation
- D - Durability

For NoSQL Systems, the Consistency in the CAP Theorem is part of the acronym:

- C - Consistency

- A - Availability
- P - Partition tolerance

The meanings are slightly different. In short:

- **Consistency in ACID** means that no dataset may be an invalid state or represents data which are semantically invalid after a transaction is committed ("internal consistency").
- **Consistency in CAP** means that after a transaction is executed this dataset must be updated in all replications too.