

IIT Madras
ONLINE DEGREE

Modern Application Development – I
Professor Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology Madras
SQL vs NoSQL

Hello, everyone, and welcome to this course on modern application development.

(Refer Slide Time: 00:16)




SQL vs NoSQL

So, all the discussion that we have had so far applies mostly to so called traditional relational database management systems or RDBMS, because we have been talking about table structured data. But you are very likely to have come across this term NoSQL at some point. So, what I want to do here is to sort of go over alternatives to SQL or these kinds of databases, and hopefully also make some things a bit more clear about why exactly we call something, NoSQL and what are the implications?

(Refer Slide Time: 00:47)



SQL

- Structured **Query** Language
 - Used to query databases that have structure
 - Could also be used for CSV files, spreadsheets etc.
- Closely tied to RDBMS - relational databases
 - Columns / Fields
 - Tables of data hold relationships
 - All entries in a table **must** have same set of columns
- Tabular databases 
 - Efficient indexing possible - use specified columns
 - Storage efficiency: prior knowledge of data size



So, first of all, what is SQL? It is a structured query language. It is used to query databases that have some kind of structure in them, but the query language as such is not tied to a given database, it could even be used for querying spreadsheets, for example. So, if you look at Google Sheets, there is a variant of SQL that can be used in order to make queries inside of Google Sheet.

Now, having said that, SQL is typically closely tied to RDBMSs, relational database management systems, where you have data being stored in tables with the values actually being in columns or fields of the table and there can be some tables that hold relationships between other tables. And the important point or here is, every entry in a given table must compulsorily have the same set of columns. This is the definition pretty much that it has a uniform table structure.

So, why are these kinds of tabular databases used? They are very good at storing data. You have prior knowledge of the data size, and you can also do efficient indexing. You can basically pick one column out of the different columns in the table and say, index on this. The database engine basically creates an index, and thereafter any searches on that particular column are going to be fast, faster than they would have been without an index.

(Refer Slide Time: 02:20)



Problem with tabular databases

- Structure (good? bad?)
- All rows in table must have same set of columns

Example

- Student - hostel => mess
- Student - day-scholar => gate pass for vehicle
- Table? Column for mess, column for gate pass???



Now, what are the problems with tabular databases? They are structured, which is usually good, but can also be bad. As an example, let us consider that you are trying to create a database to hold student information. And at least in IIT, what happens is a student could be either someone who stays in the hostel in which case you need to know which mess they go to, and what are their mess fees, have they registered and so on or they could be a day scholar, in which case, they are probably coming in from outside using their own vehicle, and in which case, they need to have a gate pass for the vehicle.

Now, in a regular tabular database, the student table would need to have one column, which indicates the mess and another column, which indicates the gate pass. Now, let us say that half the students are hostellers, half are day scholars, I have half the entries in the mess column as null, and half the entries in the gate pass column as null.

So, is this really an efficient use of space? Obviously not? But what can you do about it? The whole idea of a tabular database means that I have to have the same set of columns for every entity. So, this is sort of what got people thinking about how to store data, other alternatives.

(Refer Slide Time: 03:41)

Alternate ways to store: Document databases

- Free-form (unstructured) documents
 - Typically JSON encoded
 - Still structured, but each document has own structure
- Examples:
 - MongoDB
 - Amazon DocumentDB

JSON

```
1 {
2   "year": 2013,
3   "title": "Fargo Is Down, Or Else!",
4   "info": {
5     "director": "Miles Smith", "Bob Jones",
6     "release_date": "2013-01-18T00:00:00Z",
7     "rating": 6.2,
8     "genres": ["Comedy", "Drama"],
9     "page_url": "http://ia.media-amazon.com/images/M/MediaMarket/5701A7LBRN00M",
10    "plot": "A rock band plays their music at high volumes, annoying the neighbors.",
11    "actors": ["David Mathewson", "Jonathan G. Hoff"]
12  },
13 }
14
15 {
16   "year": 2015,
17   "title": "The Big New Movie",
18   "info": {
19     "plot": "Nothing happens at all.",
20     "rating": 0
21  },
22 }
23 }
```



And a number of alternatives have been proposed. So, one of them, for example, is a so called document database. So, document databases say that a document is essentially some free form information, but with structure. So, it is not that it is unstructured. On the right here, I have an example of a possible document. This is basically from Amazon's example, for a document database, and there are two, it is basically a JSON file or a JSON object over here.

It has two entries. So, this is one entry and this is the second entry. So, the year is common to both of them, the title is also common to both of them, and in fact, the information is also common. But within the info we find that plot is, yes, this is also common to both of them, the rating also happens to be common. But what are all these other entries? One of them has it, the other does not.

Now, a document database basically says that this is perfectly fine. I can have different entries being stored as documents inside this database, and those documents are essentially JSON objects, which means that any search in it will ultimately return a JSON object corresponding to a particular document.

How exactly do I index on it? That is a different story. You would probably say maybe you can index on the document.info.rating and then it would go through and find out all the documents that have an entry for document.info.rating and index on those, and other things that do not even have that information would not even show up in that index.

So, that is pretty much all that it says. You are storing data similar to what you would in the other structured database, with the difference being that you no longer have tables of fixed

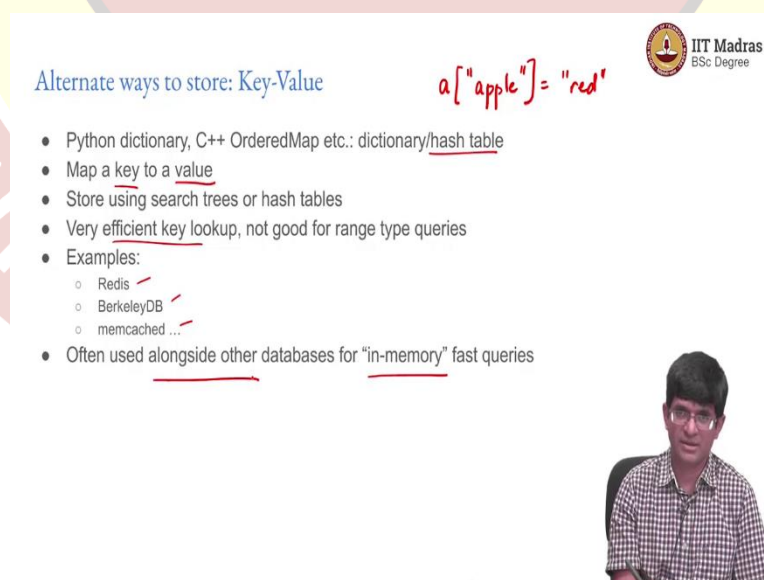
sizes, you are storing arbitrary documents. But how do your searches become efficient primarily, by the way that you construct indexes. And just like you could construct an index in the previous case over here too you can take one particular key in the JSON object and decide to construct an index on that if you wanted to make the searches fast.

So, these are actually very popular these days. MongoDB is probably one that you might have heard of, at some point. It is also probably one of the most popular in terms of document database, there are alternatives, and Amazon provides something called DocumentDB for this.

You will notice, by the way that Amazon provides alternatives for every kind of database I mean, because that is their job. I mean, the Amazon web services is ultimately trying to provide solutions for whatever it is that you are looking for. So, MongoDB is an example of a document database. You can see where it differs from a traditional tabular database.

And, the advantage over here is, let us say, going back to the previous case of students I would have a student.hosteller.mess or student.dayscholar.gatepass and they will be completely independent of each other. One student would have a hosteller substructure, the other would have a day scholar substructure, and the corresponding entries would be available there.

(Refer Slide Time: 07:19)



Alternate ways to store: Key-Value

$a["apple"] = "red"$

- Python dictionary, C++ OrderedMap etc.: dictionary/hash table
- Map a key to a value
- Store using search trees or hash tables
- Very efficient key lookup, not good for range type queries
- Examples:
 - Redis
 - BerkeleyDB
 - memcached ...
- Often used alongside other databases for "in-memory" fast queries

IIT Madras
BSc Degree

Now, there are other ways of storing data one is called key value. And in Python, you have come across this notion of dictionaries. And what is a dictionary? It basically means that, I can sort of have an array where I can say, for example, something like `a["apple"]` and I could

give it a value red. Now what is happening is that this apple is being taken as an index, in order to find some particular location, and in that location, I store the value red.

Now, one way that this is done is actually by using something called a Hash table that we already discussed earlier. There are other ways, ultimately, all that you need to do is to be able to search for a particular location, and then put something there. And that search could also be done using other techniques, such as binary search trees.

So, for example, the C++ `OrderedMap` is actually implemented as a form of a binary search tree, not as a hash table directly, but the point is it is efficient at implementation and search. It maps a key to a value and they are very efficient at key lookup, but not very good for range type queries. Once again, you might have come across this term called Redis. Redis, is usually used as a very efficient key value store. There is also memcached, which is used for something similar, the BerkeleyDB is something much older, but again, has a similar kind of behavior.

Now, one thing that is usually done over here is most of these are implemented in so called in-memory form. Meaning that, you do not really want these to be things that are stored into external disk. They really have their value when they are sitting completely in the memory of the system, and can respond to queries very fast.

So, for example, something like, if I just want to have a list of students and say, have they finished, have they completed a course or have they finished enrolling? And I retain that in Redis, then the query becomes instantaneous, almost. I give the student ID, and it tells me yes or no whether they have registered. I do not need to go to the main database and look up something. It is much faster. But the problem is, it is only a lookup based on an exact query

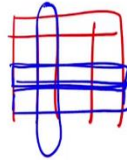
So, if I have that kind of thing, they are very fast, much faster than any other kind of database that you can think of, but they have sort of that limited use case. They are very often used alongside other databases. So, that is what you will notice in practice.

(Refer Slide Time: 09:47)



Alternate ways to store: Column stores

- Traditional relational DBs store all values of a row together on disk
 - Retrieving all entries of a given row very fast
- Instead store all entries in a column together
 - Retrieve all values of a given attribute (age, place of birth, ...) very fast
- Examples:
 - Cassandra
 - HBase...



There is something called a columnar store. And the idea here is that traditional databases sort of if I had a table I would take all the entries corresponding to one row and store it together. So, this entire thing would form one block somewhere on disk. This next one would form another block somewhere, which makes it easy to sort of retrieve all the entries corresponding to a given row.

But let us say that I have a different kind of query structure where I want to go and say, can you give me all the people born on a given date. In which case, what I am looking for is basically all the entries corresponding to a given column. Now, there are databases that sort of specialize in that. They just change the way that data is stored, so that they are better at answering these kinds of queries. Those are called column stores.

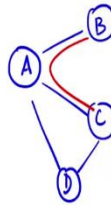
There are variants on this otherwise called also wide column stores where a large amount of data is stored inside a single column. And if you actually need to go inside, you need to go and look up the entries inside the column and figure out what is it that actually matches with your query.

(Refer Slide Time: 11:01)



Alternate ways to store: Graphs

- Friend-of-a-friend, social networks, maps: graph oriented relationships
- Different degrees (number of outgoing edges), weights of edges, nodes etc.
- Path-finding more important than just search
 - Connections, knowledge discovery
- Examples:
 - Neo4J
 - Amazon Neptune



And finally, we have this other notion of something called a graph. So, the idea of a graph is easy to explain. Let us say, there is a person A, and person A knows B, and C, and D. C and D also know each other, but none of them know B. This is an example of what Facebook is. It is basically the friend of a friend network or the social network that we are talking about.

There are some people who know lots of others, there are some who do not know many other people. But now I can sort of try and form a relationship. How do I get from B to C? And the answer is, go through A. So, if B and C want to talk to each other, maybe they can ask A for an introduction. So, those of you who have a LinkedIn account, for example, might have noticed that this is a third-degree connection or a second-degree connection. Ask so and so to introduce you, and so on. This is exactly what it is doing.

It builds up a graph internally, which has all of these connections and then based on that it is able to say, if B wants to talk to C, the best way, rather than just sending C an email and hoping that C will respond is to ask A, can you introduce me and then C is more likely to respond. So, in these kinds of graph structures, the kind of queries that you are interested in are not really give me all people with the name starting with A, instead, it is, what is the fastest way to get from B to C.

It could also be a map, a world map, and you want to find out the shortest path to get from one place to another or the cheapest set of flights that will get you to London. So, things of that sort are graph-oriented queries, and there are databases that sort of specialize in that. I mean, there is something called Neo4J and there are other databases as well, which come up in different contexts that sort of specialize in this kind of data storage and retrieval.

(Refer Slide Time: 13:06)



Alternative ways to store: Time Series Databases

- Very application specific: store some metric or values as function of time
- Used for log analysis, performance analysis, monitoring
- Queries:
 - How many hits between T1 and T2?
 - Average number of requests per second?
 - Country from where maximum requests came in past 7 days?
- Typical RDBMS completely unsuitable - same for most alternatives
- Examples:
 - RRDTool
 - InfluxDB
 - Prometheus
- Search: elasticsearch, grafana,...



And one last thing is so called, Time Series Databases, Now, these are very application specific, they are typically used to store some metric or values as a function of time. They are useful for things like log file analysis, performance monitoring, and so on. And the kind of queries that you are likely to face over here will be something like, how many hits were there to this website between, let us say, January and March of this year?

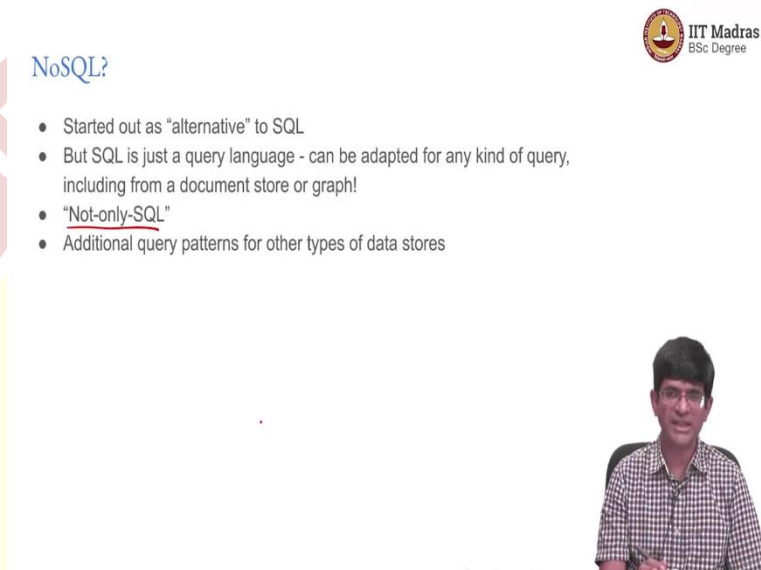
So, it should be able to do a very fast time-based query or it might be that, you know, what is the average number of requests per second, so time sort of becomes something fundamental. The problem is if you store every single query and just sort of go back and say I will put everything into this huge B-Tree, and then I just have to search through it, because time after all is ordered.

I mean, there is a clear sense of order over there, that is not very efficient for this kind of structure. There are better ways of storing it, and that is where the so called TSDB, the time series database shines. There are many examples of this. And part of the thing about TSDB is, that you may not want to store data indefinitely.

Let us say after one year, I do not really want to store every single thing that happened, every single query that came on my website, I just want an aggregate. I want to know how many queries were there in December, rather than on December 13, between 1 PM and 2 PM. So things like RRD tools Round Robin Database tool that was more or less the first one that you started applying these concepts on a large scale.

But nowadays, there are many things such as influxDB, Prometheus, and so on, which are coupled with search engines, elasticsearch, grafana various things that you might come across at various points, whose main job is to sort of help with analytics, so they can sort of take in logfile data and construct time series databases out of it, and then allow you to have very interesting queries on your data that come up with, good predictors of what is likely to happen or ways by which you can optimize your application better.

(Refer Slide Time: 15:20)



The slide is titled "NoSQL?" in blue text. It features a list of four bullet points: "Started out as 'alternative' to SQL", "But SQL is just a query language - can be adapted for any kind of query, including from a document store or graph!", "'Not-only-SQL'", and "Additional query patterns for other types of data stores". In the top right corner, there is a logo for "IIT Madras BSc Degree". In the bottom right corner, there is a small video inset showing a man with glasses and a checkered shirt speaking.

- Started out as "alternative" to SQL
- But SQL is just a query language - can be adapted for any kind of query, including from a document store or graph!
- "Not-only-SQL"
- Additional query patterns for other types of data stores

So, after all of this. Now, clearly those of you who have read more about these things, you would have come across this term, NoSQL, in the context of most of these alternative data storage mechanisms that I mentioned earlier. Now, NoSQL, what exactly is it in the first place? It started out as an alternative to SQL.

So, it was pretty much when they came out with document databases, they said, look, we are able to store data, but we do not need that tabular structure, so this is not an SQL database, this is a NoSQL database. But SQL is just a query language. It can be adapted for any kind of query. So, including from a document database.

So, what you will find these days is that even for something like MongoDB, the query structure that is used is very similar to what is used in SQL. You have select * and various kinds of things of that sort, which are clearly inspired by SQL, if not directly, a derivative of that. So nowadays, you see that, the term is more appropriately described as not only SQL.

What they are trying to convey here is that this is not limited to the kind of SQL queries that are possible on a RDBMS, and you do not in particular, need to worry about things like inner

joins, outer joins, and various other ways of constructing a query. What you should be looking at is, are there other variants of the query language, that are more efficient at looking up things in the particular data store that we are using here?

So, NoSQL, and I want to emphasize this, it is not that it is a, it is not sort of saying this is not SQL or that it is sort of violating any of the principles of SQL or that you it is not an RDBMS. Generally speaking, yes, that is true, but that is not the point. NoSQL is just sort of saying that something which is described as a NoSQL database, usually has something different in the way the storing data, which means that using the traditional SQL type of queries, may not be the best way to handle it.

(Refer Slide Time: 17:33)

A word on ACID

- Transaction: core principle of database
- ACID:
 - Atomic
 - **Consistent**
 - Isolated
 - Durable
- Many NoSQL databases sacrifice some part of ACID (example: eventual consistency instead of consistency) for performance
- But there can be ACID compliant NoSQL databases as well...

And one thing, which again, comes up in this context, is there is this notion called ACID, which once again, you might have come across in the database course, and it essentially relates to one of the core principles of databases, which is what is called a transaction. And a transaction could be something like create a new user entry. So, what does that mean? It means basically, provide space in the database, put in the user's name, ID, address, phone number, whatever all of that information, and make sure it has got stored into the database.

And ACID essentially refers to four of the principles that are there, that a particular operation such as this should be atomic. Either the entire user entity got created or the database was left unchanged. That it is consistent, meaning that you cannot have two copies of the database having different information at any given point in time.

Isolated, one series of transactions should not sort of interfere with another series of transactions, which is happening. The result of a transaction should be independent of whether something else happened in between or not. And finally, durable, also to indicate that it should actually get saved to permanent storage at some point.

Now, in all of these, this consistent part of it is probably the hardest one for a lot of databases to manage, especially, when you want to start scaling. That is when you want to increase the number of entries or increase the number of servers that are going to answer a query. And what happens is that many NoSQL databases sacrifice some part of this ACID. For example, they use something called eventual consistency, instead of direct consistency in order to get better performance.

That is not to say that, NoSQL means, it is not ACID compliant. There can be ACID compliant, NoSQL databases. In other words, I can take a document store and say that, I still need to have the ACID properties on it. There is nothing preventing that from happening. So, it is not that a NoSQL database necessarily violates acid or that anything that violates acid is a NoSQL database, you need to be a little careful about judging what is or is not a NoSQL database.

Unfortunately, it also suffers from a little bit of buzzword hype. Meaning that people sort of want to call something a NoSQL database just so that people will look at it. That is what is more important is, what does it actually give you, which is beneficial?

(Refer Slide Time: 20:08)

Why not ACID?

- Consistency hard to meet: especially when scaling / distributing
- Eventual consistency easier to meet
- Example:
 - A (located in India) and B (located in the US) both add C as a friend on Facebook
 - Order of adding does not matter!
 - Temporarily seeing C in A's list but not B, or B's list but not A - not a catastrophe (?)
- Financial transactions **absolutely require** ACID
 - Consistency is paramount - even a split second of inconsistent data can cause problems



So, why do people even want to sort of break the ACID rules? Because after all, they sound like a good idea. The main reason is that consistency is hard to meet, especially, when you want to scale up or scale to multiple servers. And there is another principle called eventual consistency, which is actually much easier to meet. So, to take an example, let us say that A and B are two people one is located in India, the other is located in the US and they both add C, as a friend on Facebook, the order in which it actually gets reflected in Facebook's database does not matter too much.

So, it might be that for a short amount of time, someone looking at the page would see C is a friend of A, but does not show up on B's friend list, but somebody else might see that, C shows up on B's friend lists, but not on A's. After some amount of time, C will show up on both that is what eventual consistency is. It does not sort of guarantee how fast or there may be guarantees on within so much time, but it is definitely not going to be instantaneous.

And the whole idea is that, you know, this is, in general, not a catastrophe. Of course, nowadays, you cannot say for sure. People might get upset that they were on one person's friend list, but not the other, but that is not what eventual consistency is trying to solve.

Now, on the other hand, a financial transaction where I debit 100 rupees from A's account, and credit it to B's account, they have to be absolutely ACID-compliant. The consistency is paramount over there. At no event, should I be able to look into the database and see that 100 rupees was deducted from A but not added to B. It has to be somewhere. So, that thing has to be very clear. And especially financial transactions, therefore, never really go for things like NoSQL. On the other hand, a lot of web applications these days do not require this kind of consistency. And that is why NoSQL is becoming popular now.

(Refer Slide Time: 22:12)



A word on storage

- In-memory:
 - Fast
 - Doesn't scale across machines
- Disk
 - Different data structures, organization needed



Now, a quick word on how data actually gets stored. And this is that, as far as possible you would actually like to store data completely in-memory. Meaning that in the RAM of the system. It is obviously going to be fast, but does not really scale across multiple machines for the simple reason that if I have different machines, they do not share the same physical memory, so I cannot replicate memory as easily as I can let them both share a disk for example. But now, if I want to store data out to a disk, remember, the high latency is associated with disk. This means that, I need to change the data structures and organize my data differently.