FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Normalisation*? ☆☆

Topics: SQL Databases

Answer:

Normalization is basically to design a database schema such that **duplicate and redundant data is avoided**. If the same information is repeated in multiple places in the database, there is the risk that it is updated in one place but not the other, leading to data corruption.

There is a number of normalization levels from 1. normal form through 5. normal form. Each normal form describes how to get rid of some specific problem.

By having a database with normalization errors, you open the risk of getting invalid or corrupt data into the database. Since data "lives forever" it is very hard to get rid of corrupt data when first it has entered the database.

Q2: 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

Q3: What is the difference between *Data Definition Language (DDL)* and *Data Manipulation Language (DML)*? ☆☆

Topics: MySQL SQL T-SQL Databases

Answer:

- **Data definition language (DDL)** commands are the commands which are used to define the database. **CREATE**, **ALTER**, **DROP** and **TRUNCATE** are some common DDL commands.

- **Data manipulation language (DML)** commands are commands which are used for manipulation or modification of data. **INSERT**, **UPDATE** and **DELETE** are some common DML commands.

Q4: What Is ACID Property Of A System? ☆☆☆

Topics: Software Architecture Databases

Answer:

ACID is a acronym which is commonly used to define the properties of a relational database system, it stand for following terms

- **Atomicity** - This property guarantees that if one part of the transaction fails, the entire transaction will fail, and the database state will be left unchanged.
- **Consistency** - This property ensures that any transaction will bring the database from one valid state to another.
- **Isolation** - This property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially.
- **Durable** - means that once a transaction has been committed, it will remain so, even in the event of power loss.

Q5: What are the difference between *Clustered* and a *Non-clustered* index? ☆☆☆

Topics: SQL Databases

Answer:

- With a **Clustered** index the rows are stored **physically** on the disk in the same order as the index. Therefore, there can be only one clustered index. A clustered index means you are telling the database to store close values actually close to one another on the disk.
- With a **Non Clustered** index there is a second list that has **pointers** to the physical rows. You can have many non clustered indices, although each new index will increase the time it takes to write new records.
- It is generally *faster to read* from a **clustered** index if you want to get back all the columns. You do not have to go first to the index and then to the table.
- *Writing* to a table with a **clustered** index can be *slower*, if there is a need to rearrange the data.

Q6: What's the difference between a *Primary Key* and a *Unique Key*? ☆☆☆

Topics: SQL Databases

Answer:

Primary Key:

- There can only be one primary key in a table
- In some DBMS it cannot be `NULL` - e.g. MySQL adds `NOT NULL`

- Primary Key is a *unique* key identifier of the record
- Primary key can be created on multiple columns (composite primary key)

Unique Key:

- Can be more than one *unique* key in one table
- Unique key can have `NULL` values
- It can be a candidate key
- Unique key can be `NULL` ; multiple rows can have `NULL` values and therefore may not be considered "unique"

Difference between Primary Key and Unique key

- **1. Behavior:** Primary Key is used to identify a row (record) in a table, whereas Unique-key is to prevent duplicate values in a column (with the exception of a null entry).
- **2. Indexing:** By default SQL-engine creates Clustered Index on primary-key if not exists and Non-Clustered Index on Unique-key.
- **3. Nullability:** Primary key does not include Null values, whereas Unique-key can.
- **4. Existence:** A table can have at most one primary key, but can have multiple Unique-key.
- **5. Modifiability:** You can’t change or delete primary values, but Unique-key values can.

Q7: What is Denormalization? ☆☆☆

Topics: SQL Databases

Answer:

It is the process of improving the performance of the database by *adding* redundant data.

Q8: Define ACID Properties ☆☆☆

Topics: SQL Databases

Answer:

- **Atomicity**: It ensures all-or-none rule for database modifications.
- **Consistency**: Data values are consistent across the database.
- **Isolation**: Two transactions are said to be independent of one another.
- **Durability**: Data is not lost even at the time of server failure.

Q9: How a database index can help performance? ☆☆☆

Topics: SQL Databases

Answer:

The whole point of having an index is to speed up search queries by essentially cutting down the number of records/rows in a table that need to be examined. An index is a data structure (most commonly a B- tree) that stores the values for a specific column in a table.

Q10: What is *Optimistic Locking*? ☆☆☆

Topics: Entity Framework Databases

Answer:

Optimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.

If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

Q11: 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.

Q12: 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".

Q13: What Is Sharding? ☆☆☆☆

Topics: Software Architecture Databases

Answer:

Sharding is a architectural approach that distributes a single logical database system into a cluster of machines. Sharding is *Horizontal partitioning* design scheme. In this database design rows of a database table are stored separately, instead of splitting into columns (like in *normalization* and *vertical partitioning*). Each partition is called as a shard, which can be independently located on a separate database server or physical location.

Sharding makes a database system highly scalable. The total number of rows in each table in each database is reduced since the tables are divided and distributed into multiple servers. This reduces the index size, which generally means improved search performance. The most common approach for creating shards is by the use of consistent hashing of a unique id in application (e.g. user id).

The downsides of sharding:

- It requires application to be aware of the data location.
- Any addition or deletion of nodes from system will require some rebalance to be done in the system.
- If you require lot of cross node join queries then your performance will be really bad. Therefore, knowing how the data will be used for querying becomes really important.
- A wrong sharding logic may result in worse performance. Therefore make sure you shard based on the application need.

Q14: 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.

Q15: Explain the difference between *Exclusive Lock* and *Update Lock* ☆☆☆☆

Topics: SQL Databases

Answer:

- In case of **Exclusive lock**, no other lock can be acquired on that row or table. Every process has to wait until the process which holds the lock releases it.
- In case of **Update lock**, while reading the row or a record, you can have any other lock associated with that row or record. In case of updating the record, update lock changes itself to an exclusive lock and no other process can obtain a lock on that row until the lock is released.

Q16: How does *B-trees Index* work? ☆☆☆☆

Topics: Databases SQL

Answer:

The main reason for the existence of **B-Tree Indexes** is to better utilize the behaviour of devices that read and write large chunks of data. Two properties are important to make the B-Tree better than binary trees when data has to be stored on disk:

- Access to disk is really slow (compared to memory or caches, random access to data on disk is orders of magnitude slower); and
- Every single read causes a whole sector to be loaded from the drive - assuming a sector size of 4K, this means 1000 integers, or tens of some larger objects you're storing.

Hence, we can use the pros of the second fact, while also minimizing the cons - i.e. number of disk accesses.

So, instead of just storing a single number in every node that tells us if we should continue to the left or to the right, we can create a bigger index that tells us if we should continue to the first 1/100, to the second or to the 99-th (imagine books in a library sorted by their first letter, then by the second, and so on). As long as all this data fits on a single sector, it will be loaded anyway, so we might as well use it completely.

This results in roughly $\log_b N$ lookups, where N is the number of records. This number, while asymptotically the same as $\log_2 N$, is actually a few times smaller with large enough N and b - and since we're talking about storing data to disk for use in databases, etc., the amount of data is usually large enough to justify this.

Q17: What is the *cost* of having a database *index*? ☆☆☆☆

Topics: Databases SQL

Answer:

It takes up space – and the larger your table, the larger your index. Another performance hit with indexes is the fact that whenever you add, delete, or update rows in the corresponding table, the same operations will have to be done to your index. Remember that an index needs to contain the same up-to-the-minute data as whatever is in the table column(s) that the index covers.

As a general rule, an index should only be created on a table if the **data** in the indexed column will be **queried frequently**.

Q18: 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.

Q19: 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).

Q20: How do you off load work from the Database? ☆☆☆☆

Topics: Databases Software Architecture

Answer:

Here is a list of standard options.

- **Optimize the access** to the database to only do what you need, efficiently. A good DBA can help here a lot. This is a basic step that most companies do.
- **Cache data away from the database** using something like memcached. This is usually done at the application layer, and is highly effective. Virtually every competent website should do this.
- More ambitiously, maintain **read-only copies of the database**, and direct queries there when possible. On the database side the necessary technology is called "replication" and the read-only copies are often also backups for failover from the main database. If you're doing a million dynamic pages per hour, odds are that you are doing this, or have thought about it.
- Buy really, really **expensive hardware** for the database. I know that PayPal did this as of 4 years ago, and changing their architecture would have been difficult so they possibly still are.
- **Shard the database** into multiple pieces with ranges of data. This is a very intrusive change into application design. A well-known example of a company that does this is eBay.
- Try to use a database that **scales onto multiple machines**. Oracle RAC scales onto clusters, but doesn't let you distribute data widely. Other offerings exist that are supposed to be easier to distribute, including Microsoft's SQL Azure and FathomDB. I have not used those offerings and don't know how well they work. I suspect better than nothing, but I doubt they scale horizontally that well.
- Relational databases generally try to provide ACID guarantees. But the CAP theorem makes it very difficult to do that in a distributed system, particularly while letting you do things like join data. Therefore people have come up with many **NoSQL alternatives** that explicitly offer weaker guarantees and avoid problematic operations in return for **fully distributed scalability**. Well-known examples of companies that use scalable NoSQL data stores include Google, Facebook and Twitter.

FullStack.Cafe - Kill Your Tech Interview

Q1: What Does Eventually Consistent Mean? ☆☆☆☆☆

Topics: Software Architecture Databases

Answer:

Unlike relational database property of Strict consistency, *eventual consistency* property of a system ensures that any transaction will eventually (not immediately) bring the database from one valid state to another. This means there can be intermediate states that are not consistent between multiple nodes.

Eventually consistent systems are useful at scenarios where absolute consistency is not critical. For example in case of Twitter status update, if some users of the system do not see the latest status from a particular user its may not be very devastating for system.

Eventually consistent systems can not be used for use cases where absolute/strict consistency is required. For example a banking transactions system can not be using eventual consistency since it must consistently have the state of a transaction at any point of time. Your account balance should not show different amount if accessed from different ATM machines.

Q2: What is *Optimistic Locking* and *Pessimistic Locking*? ☆☆☆☆☆

Topics: SQL Databases

Answer:

In Short:

- **Optimistic** assumes that nothing's going to change while you're reading it.
- **Pessimistic** assumes that something will and so locks it.

Detailed answer:

- **Optimistic Locking** is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version *hasn't changed before you write* the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.

  If the record is *dirty* (i.e. different version to yours) you abort the transaction and the user can re-start it.

  This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation, the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

- **Pessimistic Locking** is when you *lock the record* for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid *Deadlocks*. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.

The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time.

Q3: How does database *Indexing* work? ☆☆☆☆☆

Topics: SQL Databases

Answer:

Indexing is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure (stored on the disc) which holds the field value, and a pointer to the record it relates to. This index structure is then sorted, allowing Binary Searches to be performed on it, which has log2 N block accesses. Also since the data is sorted given a non-key field, the rest of the table doesn't need to be searched for duplicate values, once a higher value is found.

The whole point of having an index is to **speed up search queries** by essentially cutting down the number of records/rows in a table that need to be examined. An index is a data structure (most commonly a B- tree or hash table) that stores the values for a specific column in a table.

Given the nature of a binary search, the **cardinality or uniqueness** of the data is important. Indexing on a field with a cardinality of 2 would split the data in half, whereas a cardinality of 1,000 would return approximately 1,000 records.

Q4: Name some disadvantages of a *Hash index* ☆☆☆☆☆

Topics: SQL Databases

Answer:

- **Hash** tables are *not sorted* data structures, and there are many types of queries that hash indexes can not even help with.

For instance, suppose you want to find out all of the employees who are less than 40 years old. How could you do that with a hash table index? Well, it's not possible because a hash table is only good for looking up key-value pairs – which means queries that check just for *equality*.

Q5: What are some *other* types of Indexes (vs B-Trees)? ☆☆☆☆☆

Topics: SQL Databases

Answer:

- Indexes that use a **R-tree** data structure are commonly used to help with *spatial* problems. For instance, a query like "Find all of the Starbucks within 2 kilometers of me" would be the type of query that could show enhanced performance if the database table uses a R-tree index.
- Another type of index is a **bitmap index**, which works well on columns that contain Boolean values (like true and false), but many instances of those values – basic columns with low selectivity.

Q6: 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

Q7: 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.

Q8: How do you make *schema changes* to a live database without downtime? ☆☆☆☆☆

Topics: Databases PostgreSQL MySQL

Problem:

Let's say I have a PostgreSQL/MySQL database with a table including various user data like email addresses etc, all associated with specific users. If I wanted to move the email addresses to a new dedicated table, I'd have to change the schema and then migrate the email data across to the new table. How could this be done without stopping writes to the original table?

Solution:

Follow these steps:

1. Create the new structure in parallel
2. Start writing to both structures
3. Migrate old data to the new structure
4. Only write and read new structure
5. Delete old columns

As for **step 3**, use something like this (in one transaction):

Insert what is not there yet:

```
INSERT INTO new_tbl (old_id, data)
SELECT old_id, data
FROM   old_tbl
WHERE  NOT EXISTS (SELECT * FROM new_tbl WHERE new_tbl.old_id = old_tbl.old_id);
```

Update what has changed in the meantime:

```
UPDATE new_tbl
SET    data  = old.data
USING  old_tbl
WHERE  new_tbl.old_id = old_tbl.old_id
AND    new_tbl.data IS DISTINCT FROM old_tbl.data;
```

New data will not be touched, because it is identical in both places.

Q9: Why you should never use GUIDs as part of *clustered index*? ☆☆☆☆☆

Topics: Databases MySQL PostgreSQL T-SQL

Answer:

GUIDs are not *sequential*, thus if they are part of clustered index, every time you insert new record, database would need to rearrange all its memory pages to find the right place for insertion, in case with int(bigint) auto-increment, it would be just last page.

1. MySQL - primary keys are clustered, with no option to change behavior - the recomendation is not to use GUIDs at all here
2. Postgres, MS-SQL - you can make GUID as primary key unclustered, and use another field as clustered index, for example autoincrement int.

Q10: What is the difference between *B-Tree*, *R-Tree* and *Hash* indexing? ☆☆☆☆☆

Topics: SQL Databases

Answer:

BTree (in fact B*Tree) is an efficient ordered key-value map. Meaning:

- given the key, a BTree index can quickly find a record,
- a BTree can be scanned in order.
- it's also easy to fetch all the keys (and records) within a range.

> **e.g.** "all events between 9am and 5pm", "last names starting with 'R'"

RTree is a `spatial index` which means that it can quickly identify `close` values in 2 or more dimensions. It's used in geographic databases for queries such as:

> all points within X meters from (x,y)

Hash is an unordered key-value map. It's even more efficient than a BTree: `O(1)` instead of `O(log n)`. But it doesn't have any concept of order so it can't be used for sort operations or to fetch ranges.

Q11: What is *Index Cardinality* and why does it matter? ☆☆☆☆☆

Topics: Databases MongoDB SQL

Answer:

The **Index Cardinality** refers to how many possible values there are for a field.

Imagine we have a collection of all humans on earth with the following field:

```
- "sex" // 99.9% of the time "male" or "female", but string nonetheless (not binary)
- "name"
- "phone"
- "email"
```

The field `sex` only has two possible values. It has a very **low cardinality**. Other fields such as `names, usernames, phone numbers, emails`, etc. will have a more unique value for every document in the collection, which is considered **high cardinality**.

It's a rule of thumb to create indexes on `high-cardinality` keys or put `high-cardinality` keys *first* in the compound index.

Why?

The greater the cardinality of a field the more helpful an index will be, **because indexes narrow the search space, making it a much smaller set**.

Suppose you have an index on `sex` and are looking for men named John. You would only narrow down the resulting space by approximately `%50` if you indexed by `sex` first. Conversely, if you indexed by `name`, you would immediately narrow down the result set to a minute fraction of users named John, and then you would refer to those documents to check the gender.