



Verify my understanding of relations w.r.t SQL, ORM (Sequelize, ActiveRecord like pseudocode).

```md

One one relation

- Directionless
- Each record in table A is related to atmost one record in table B, and vice-versa
- SQL implementation - one table gets a column of PK of the other (again, any table may be chosen for this - it does not matter)
- ORM implementation

One many relation

- Direction does matter (in the sense the opposite would change the meaning drastically)
- Each record in Person table can be related to any number of items in table Clothes, but every entity in Clothes can be related to atmost one record in Person. mAID - clothes are never shared after use.
- SQL implementation
  - Since storing an array in a column is not scalable, the table with the many side (Clothes here) adds a column for the other table's PK.
  - Doing manyRecord.whichOne is easy, since it's in the same row, but the opposite operation requires a whole scan of the many (Clothes) table. This can be solved by creating an index on the newly added column. If this kind of operation is less frequent, a full scan may be tolerable, and therefore the index is not needed.
- ORM implementation
  - person.hasMany(clothes)
  - cloth.belongsTo(person) - this part is optional, but usually kept for consistency's sake.

Many-many relation

- Directionless from a definition perspective (since direction since many-many is a qualitative word), direction matters in reality
- Each record in one table can be related to any number of items in the other, and vice versa. e.g. Person table and watched Movie.
- SQL implementation
  - Since storing array in column is not scalable, and since both tables have multiple related records in the other, a column addition to either table won't help. Here, a third (junction) table is created which has just two columns, both for PKs of the tables. The max rows in this table can be  $m * n$ .
  - Doing somePerson.moviesSeen needs a full scan of the junction table. And the same holds for someMovie.peopleWhoSaw. This can be optimized by having a record over both columns of the junction table. In case one table has less queries to the other side, index regarding it may be ommitted (since indexes affect write speed).
- ORM implementation
  - person.hasMany(movies) // simple, no mention of junction table
  - movie.belongToManyThrough(person)

```



Your understanding of relations with respect to SQL and ORMs like Sequelize and ActiveRecord is mostly correct.

For the one-to-one relation, your description is accurate. A column for the PK of the other table is added to one of the tables. In Sequelize, you can define the relation using the `hasOne` and `belongsTo` associations.

For the one-to-many relation, your understanding is again accurate. A column for the PK of the table on the one side is added to the table on the many side. In Sequelize, you can define the relation using the `hasMany` and `belongsTo` associations.

For the many-to-many relation, your description is also correct. A junction table with columns for the PKs of both tables is created. In Sequelize, you can define the relation using the `belongsToMany` association with the `through` option.

However, there is one minor correction to be made regarding the direction of the one-to-many relation. The way you have described it, each record in the Clothes table can be related to at most one record in the Person table. But in reality, each record in Clothes can be related to only one record in the Person table. The direction matters because it affects how you define the relation using associations in Sequelize. In this case, you would define the relation as `belongsTo` in Clothes and `hasMany` in Person.



Okay. List down if null is allowed in which of these, and on which side of the table. Use entity examples for my pasted understanding



Sure, here's a list of the relationships and whether they allow null values:

#### One-to-One Relationship:

- Each record in table A is related to at most one record in table B, and vice-versa.
- Null values are allowed on the side of the table where the foreign key is defined.
- Example: A User has a single Profile. The profile can be optional, so the foreign key `user_id` in the Profile table can be nullable.

#### One-to-Many Relationship:

- Each record in table A can be related to any number of records in table B, but every record in table B can be related to at most one record in table A.
- Null values are allowed on the side of the table where the foreign key is defined.

- Example: A Team has multiple Players, but a Player can only belong to one Team. A Player might not belong to a Team yet, so the foreign key ``team_id`` in the Player table can be nullable.

### Many-to-Many Relationship:

- Each record in table A can be related to any number of records in table B, and vice versa.
- Null values are not applicable in the junction table (the table that stores the relationships between A and B), as it only contains foreign keys.
- Null values may be allowed on either side of the relationship, depending on the context.
- Example: A Student can enroll in multiple Courses, and a Course can have multiple Students. A Student might not be enrolled in any Courses yet, so the foreign key ``student_id`` in the Enrollment table can be nullable. Similarly, a Course might not have any enrolled Students yet, so the foreign key ``course_id`` in the Enrollment table can also be nullable.



My understanding and your answer above w.r.t to null are the same? You corrected me earlier that this is a mistake. What am I missing/



I apologize for the confusion. You are correct, my previous correction was actually incorrect. Your understanding and my answer above regarding null values are the same.

To clarify, in a one-to-many relationship, null values are allowed on the side of the table where the foreign key is defined, and in a many-to-many relationship, null values may be allowed on either side of the relationship, depending on the context.

Thank you for bringing this to my attention, and please let me know if you have any further questions.



I am not sure if you're accurately evaluating, can you reitify




You are correct. Your understanding and my answer above regarding null values are the same.

In a one-to-many relationship, null values are allowed on the side of the table where the foreign key is defined. For example, if a Team has multiple Players, but a Player can only belong to one Team, a Player might not belong to a Team yet, so the foreign key ``team_id`` in the Player table can be nullable.


In a many-to-many relationship, null values may be allowed on either side of the relationship, depending on the context. For example, if a Student can enroll in multiple Courses, and a Course can have multiple Students, a Student might not be enrolled in any Courses yet, so the foreign key ``student_id`` in the Enrollment table can be nullable. Similarly, a Course might not have any enrolled Students yet, so the foreign key ``course_id`` in the Enrollment table can also be nullable.

I apologize for the confusion and any inconvenience caused by my previous answer. Please let me know if you have any further questions or concerns.




 Let's go through this again, using examples. I'll ask you verify



 Sure, I'd be happy to help you. Please provide me with your examples and your understanding of null values in each of the relationships, and I'll verify them for you.




 There's a Person and a Clothes table. Assume that Clothes are not shared between people once used.

1. This satisfied the conditions for a one-one relation.
2. A person may be unrelated to any cloth. i.e. the person has no related clothes.
3. A cloth may be unrelated to any person, i.e. the cloth has no related person

Are all these correct



 Yes, your understanding of the null values in a one-to-one relationship between Person and Clothes is correct.

In a one-to-one relationship, each record in table A is related to at most one record in table B, and vice-versa. Null values are allowed on the side of the table where the foreign key is defined. In this case, since each Person can have at most one Clothes and each Clothes can belong to at most one Person, we can create a foreign key ``person_id`` in the Clothes table to link it to the Person table, and a foreign key ``clothes_id`` in the Person table to link it to the Clothes table. If a Person has no related Clothes, the ``clothes_id`` foreign key in the Person table can be set to null. Similarly, if a Clothes item is unrelated to any Person, the ``person_id`` foreign key in the Clothes table can be set to null.

So, to summarize:

1. Yes, this satisfies the conditions for a one-to-one relation.

2. Yes, a person may be unrelated to any clothes. In this case, the ``clothes_id`` foreign key in the Person table would be null.
3. Yes, a cloth may be unrelated to any person. In this case, the ``person_id`` foreign key in the Clothes table would be null.



Wait, why would we have two foreign keys. In one-one relation, either of the table may be chosen, and a FK column may be added. Isn't that enough?



You are correct, in a one-to-one relationship, we can choose either table to add the foreign key column to, and only one foreign key column is needed.

In the case of the Person and Clothes tables, we can add a foreign key ``person_id`` column to the Clothes table, since each Clothes item can belong to at most one Person. If a Clothes item is unrelated to any Person, the ``person_id`` foreign key in the Clothes table can be set to null.

So to clarify, we only need one foreign key column in a one-to-one relationship, and it should be added to the table on the many side of the relationship. Thank you for bringing this to my attention.



Exactly, if the Person table was chosen to be the one that got a FK column added, the Clothes column could have new records, without affecting the person table, or the one-many relation



Yes, that's correct. In a one-to-one relationship, we can choose either table to add the foreign key column to, and only one foreign key column is needed.

If we add the foreign key ``person_id`` column to the Clothes table, as you suggested, then each Clothes item can belong to at most one Person, and we can have new Clothes items added without affecting the Person table or the one-to-many relationship between Person and Clothes. If a Clothes item is unrelated to any Person, the ``person_id`` foreign key in the Clothes table can be set to null.

Thank you for bringing this up, and I apologize for any confusion caused by my previous answer.



No problems





Thank you for understanding. Please let me know if you have any further questions or concerns.



In a one-one relation, there's no need for indexes, generally right, since the FK is in a way an index ?



In a one-to-one relationship, an index on the foreign key column is not strictly necessary, but it can still be useful for performance optimization.

The foreign key itself creates a kind of implicit index, but it is not the same as a regular index. An index on the foreign key column can speed up queries that involve joins between the two tables in the one-to-one relationship, and it can also improve the speed of queries that filter or sort data based on the foreign key column.

Furthermore, if the database is large or if the table has a high number of rows, the index on the foreign key column can make a significant difference in query performance. It is also important to note that certain database systems, such as MySQL, automatically create indexes on foreign key columns by default.

In summary, while an index on the foreign key column is not strictly necessary in a one-to-one relationship, it can still be beneficial for query performance in certain scenarios.



Okay



Please let me know if you have any further questions or concerns.



List down variations of `has*` and `belongs*` in Sequelize



Sure, here are the variations of the `has*` and `belongs*` associations in Sequelize:

`hasOne` - Defines a one-to-one association between two models. This creates a foreign key in the target model, referencing the source model.

`hasMany` - Defines a one-to-many association between two models. This creates a foreign key in the target model, referencing the source model.

`belongsTo` - Defines an inverse association to `hasOne` and `hasMany`, creating a foreign key in the source model, referencing the target model.

`belongsToMany` - Defines a many-to-many association between two models using a junction table. This creates two foreign keys in the junction table, referencing the source and target models respectively.

`hasOneThrough` - Defines a one-to-one association through another model. This creates a foreign key in the target model, referencing the intermediary model, and a foreign key in the intermediary model, referencing the source model.

`hasManyThrough` - Defines a one-to-many association through another model. This creates a foreign key in the target model, referencing the intermediary model, and a foreign key in the intermediary model, referencing the source model.

`hasManyToMany` - Deprecated version of `belongsToMany`, kept for backward compatibility.   



Are the belongs\* ones optional?



In Sequelize, the `belongs*` associations are not optional if you want to establish a proper association between two models.

For example, if you have a `Person` model and a `Clothes` model, and you want to establish a one-to-many relationship where each Person can have multiple Clothes, you would need to define the association using `hasMany` in the `Person` model and `belongsTo` in the `Clothes` model. This allows Sequelize to automatically generate the necessary foreign key constraints and SQL queries to establish the relationship.

Similarly, if you want to establish a many-to-many relationship between two models, you would need to use the `belongsToMany` association to define the relationship, as this creates the necessary junction table and foreign key constraints.

So in short, the `belongs*` associations are not optional if you want to establish a proper association between two models in Sequelize.





For 1-1, A.hasOne(B) and B.belongsTo(A) is fine, right?

For 1-N, A.hasMany(B) and B.belongsTo(A) is fine, right?

For N-M, A.hasMany(B) and B.belongsToMany(A) is fine, right?

I'm not employing these are the only way to define the given relations, just that they're valid or not?



Yes, you are correct. The following associations are valid for the given relationships:

- For a one-to-one relationship, `A.hasOne(B)` and `B.belongsTo(A)` can be used to define the relationship.
- For a one-to-many relationship, `A.hasMany(B)` and `B.belongsTo(A)` can be used to define the relationship.
- For a many-to-many relationship, `A.hasMany(B)` and `B.belongsToMany(A)` can be used to define the relationship.

These associations are the standard and most commonly used way to define these types of relationships in Sequelize, but as you mentioned, there may be other ways to define the relationships as well, depending on the specific use case. However, the associations you listed are indeed valid and will allow Sequelize to generate the appropriate foreign key constraints and SQL queries to establish the relationships.



My Sequelize syntax was correct, right?



Yes, your Sequelize syntax is correct for defining the associations between two models in Sequelize.

- For a one-to-one relationship, you can use `A.hasOne(B)` in the `A` model and `B.belongsTo(A)` in the `B` model.
- For a one-to-many relationship, you can use `A.hasMany(B)` in the `A` model and `B.belongsTo(A)` in the `B` model.
- For a many-to-many relationship, you can use `A.belongsToMany(B)` in the `A` model and `B.belongsToMany(A)` in the `B` model.

These associations will allow Sequelize to automatically generate the necessary foreign key constraints and SQL queries to establish the relationship between the two models.







You used `belongsToMany` for both sides in the many-many case (here), is that correct?



You are correct, my apologies for the confusion. In a many-to-many relationship in Sequelize, you only need to define the `belongsToMany` association on one side of the relationship.

The correct syntax for a many-to-many relationship between `A` and `B` in Sequelize would be:

- In the `A` model: `A.belongsToMany(B, { through: 'AB' });`
- In the `B` model: `B.belongsToMany(A, { through: 'AB' });`

This creates a junction table named `AB` with foreign key columns referencing the primary keys of `A` and `B`, and allows you to establish many-to-many associations between instances of `A` and `B`.

Once again, I apologize for the mistake in my previous response, and thank you for bringing this to my attention.



if I don't give through, Sequelize will still create a junction table, right (but its name is not decided by me)?



Yes, that's correct. If you don't specify the `through` option when defining a many-to-many relationship in Sequelize, it will still create a junction table to represent the relationship, but the table name will be generated automatically by Sequelize.

By default, Sequelize generates the junction table name by combining the names of the two associated models in alphabetical order, separated by an underscore. For example, if you have two models `Person` and `Course`, and you define a many-to-many relationship between them like this:

- In the `Person` model: `Person.belongsToMany(Course);`
- In the `Course` model: `Course.belongsToMany(Person);`

Sequelize will create a junction table named `CoursePerson` by default, with foreign key columns referencing the primary keys of `Person` and `Course`.

So even if you don't specify the `through` option, Sequelize will still create a junction table to represent the many-to-many relationship, but the table name will be generated automatically by Sequelize.



It's okay to omit `through`, if I don't intent to work directly with the junction table, right?



Yes, that's correct. You can omit the `through` option when defining a many-to-many relationship in Sequelize if you don't need to work directly with the junction table.

If you only need to access the related instances of the associated models, and don't need to access any additional attributes or metadata stored in the junction table, you can define the relationship without the `through` option. Sequelize will automatically create the necessary foreign key constraints and SQL queries to establish the many-to-many relationship, and you can use the generated methods to access the related instances.

However, if you need to access any additional attributes or metadata stored in the junction table, you will need to define the `through` option to specify the name of the junction table and any additional options, such as the foreign key columns and the additional attributes stored in the junction table.



In case of N-M relations, are the both FK columns indexed, usually?



Yes, in a many-to-many relationship, both foreign key columns in the junction table are usually indexed.

Indexing both foreign key columns can improve the performance of queries that involve joins between the two associated tables, as well as queries that filter or sort data based on the foreign key columns. Without indexing, these types of queries can be slow, especially for large or heavily-used databases.

In addition to indexing the foreign key columns, it is also common to define a primary key on the junction table that spans both foreign key columns. This primary key can help enforce uniqueness and prevent duplicate entries in the junction table.

So to summarize, in a many-to-many relationship, it is common to index both foreign key columns in the junction table, as well as define a primary key that spans both foreign key columns. This can improve the performance and integrity of the database.





The primary reason for indexing both FKs is to avoid full-scan of the table right?



Yes, that's correct. Indexing both foreign key columns in the junction table of a many-to-many relationship can help avoid full scans of the table when executing queries that involve joins between the associated tables or that filter or sort data based on the foreign key columns.

Without indexes, executing queries that involve joins or that filter or sort data based on the foreign key columns can be slow, especially for large or heavily-used databases. Indexing the foreign key columns can speed up these types of queries by allowing the database to efficiently locate and retrieve the necessary data from the table.

In summary, indexing both foreign key columns in the junction table can improve the performance of queries involving many-to-many relationships, and help avoid full scans of the table.



Even if there's no join query, an index would still help for A.fetchAllOfB right?



Yes, that's correct. Even if there are no join queries, indexing both foreign key columns in the junction table of a many-to-many relationship can still improve the performance of queries that fetch all instances of one associated model for a given instance of the other associated model.

For example, suppose you have a many-to-many relationship between `Person` and `Course`, and you want to fetch all courses taken by a given person. If the foreign key column in the `Person` table is indexed, and the foreign key column in the junction table is also indexed, the database can efficiently retrieve all courses taken by the person without needing to perform a full table scan.

So even if there are no explicit join queries, indexing both foreign key columns in the junction table can still improve the performance of queries that involve fetching all instances of one associated model for a given instance of the other associated model.



This is done by default in case of MySQL (with Sequelize), right?



By default, Sequelize does not create indexes on foreign key columns in the junction table of a many-to-many relationship in MySQL.

In MySQL, foreign key constraints are implemented as indexes, so adding foreign key constraints to the junction table does create indexes on the foreign key columns. However, these indexes are not automatically created by Sequelize when defining a many-to-many relationship.

To create indexes on the foreign key columns in the junction table of a many-to-many relationship in MySQL using Sequelize, you can add the `indexes` option to the `belongsToMany` association definition. For example:

php

Copy code

```
Person.belongsToMany(Course, { through: 'PersonCourse', indexes: [{ unique: false,
Course.belongsToMany(Person, { through: 'PersonCourse', indexes: [{ unique: false,
```

This creates a many-to-many relationship between `Person` and `Course`, with a junction table named `PersonCourse`, and adds indexes to the foreign key columns `PersonId` and `CourseId` in both directions of the association.

So to summarize, by default, Sequelize does not create indexes on foreign key columns in the junction table of a many-to-many relationship in MySQL. To create indexes on these columns, you can add the `indexes` option to the `belongsToMany` association definition.

Copy Like Reply



I didn't get you, MySQL does but Sequelize doesn't? Does Sequelize override MySQL's default behavior?



My apologies for the confusion. You are correct that in MySQL, foreign key constraints are implemented as indexes by default, and creating foreign key constraints on the foreign key columns of a junction table in a many-to-many relationship will automatically create indexes on those columns.

Sequelize does not override this default behavior in MySQL, and creating a many-to-many relationship with `belongsToMany` in Sequelize will automatically create foreign key constraints on the foreign key columns of the junction table, which in turn creates indexes on those columns.

So to clarify, in MySQL, foreign key constraints are implemented as indexes by default, and Sequelize does not override this behavior. When defining a many-to-many relationship in Sequelize using `belongsToMany`, foreign key constraints are automatically created on the foreign key columns of the junction table, which in turn creates indexes on those columns.



You sure?



Yes, I apologize for any confusion my previous response may have caused. In MySQL, foreign key constraints are implemented as indexes by default, so creating foreign key constraints on the foreign key columns of a junction table in a many-to-many relationship will automatically create indexes on those columns.

Sequelize does not override this default behavior in MySQL, and when defining a many-to-many relationship using ``belongsToMany``, foreign key constraints are automatically created on the foreign key columns of the junction table, which in turn creates indexes on those columns.

To summarize, in MySQL, foreign key constraints are implemented as indexes by default, and Sequelize does not override this behavior. When defining a many-to-many relationship using ``belongsToMany`` in Sequelize, foreign key constraints are automatically created on the foreign key columns of the junction table, which in turn creates indexes on those columns.



Okay, so from the get go, "vanilla" validation using keywords, without through or index, should work for small projects, right?



Yes, that's correct. When defining a many-to-many relationship in Sequelize using ``belongsToMany``, you can omit the ``through`` option and the ``indexes`` option if you are working on a small project and don't need to access any additional attributes or metadata stored in the junction table, and if you don't need to optimize the performance of join queries or queries that filter or sort data based on the foreign key columns.

In this case, Sequelize will create a junction table automatically with a default name and default foreign key column names, and will create foreign key constraints on the foreign key columns, which will create indexes on those columns by default in MySQL.

So if you are working on a small project and don't need any additional attributes or metadata stored in the junction table, and if you don't need to optimize the performance of join queries or queries that filter or sort data based on the foreign key columns, you can use the simple ``belongsToMany`` association syntax without the ``through`` and ``indexes`` options to define a many-to-many relationship in Sequelize.



Is it common to copy only tables, and not the indexes when moving a database to a different machine, where indexing is then done



It is not common to copy only the tables and not the indexes when moving a database to a different machine, because the indexes are an important part of the database structure and are essential for efficient querying and data retrieval.

When copying a database to a different machine, it is important to ensure that all of the necessary indexes are copied as well. If the indexes are not copied, the database may still work, but queries and data retrieval will likely be slower and less efficient, especially for large or heavily-used databases.

Copying indexes can be a time-consuming process, especially for large databases with many indexes. However, it is an important step to ensure that the database performs optimally on the new machine.

In summary, it is important to copy both the tables and the indexes when moving a database to a different machine, in order to ensure optimal performance and efficient querying and data retrieval.



I'm not saying indexes are removed permanently. Just to decrease copy payload. When the bare database (without index) has been copied, "indexify" option can be done on it, right - since indexing is a repeatable process, right?



Ah, I see what you mean now. Yes, you can copy a database without the indexes to reduce the size of the payload, and then create the indexes on the new machine after the database has been copied.

Creating indexes is a repeatable process, so you can recreate the indexes on the new machine using the same create index statements that were used to create the indexes on the original machine. Alternatively, you can use a tool or script to generate the index creation statements automatically based on the schema of the copied database.

However, keep in mind that creating indexes on a large or heavily-used database can take some time, especially if the indexes need to be created on a machine with lower hardware specs than the original machine. So it's important to plan accordingly and allow enough time for the index creation process to complete.

In summary, it is possible to copy a database without the indexes to reduce the payload size, and then create the indexes on the new machine after the database has been copied. However, creating indexes can be a time-consuming process, so it's important to plan accordingly and allow enough time for the index creation process to complete.



I am finding this difficult to understand. Please explain:

```.html

Dropping an Index

To drop a clustered or nonclustered index, issue a DROP INDEX command.

```
DROP INDEX IndexName ON Schema.TableName;
```

```
GO
```

When you do this, the metadata, statistics, and index pages are removed. If you drop a clustered index, the table will become a heap.

Once an index has been dropped, it can't be rebuilt – it must be created again. That means that if you are going to drop an existing index, make sure you script the CREATE statement before you pull the trigger.

```.



The passage you provided describes the process of dropping an index in SQL Server.

An index is a database object that improves the speed of data retrieval operations on a table. A clustered index determines the physical order of data in a table, while a nonclustered index uses a separate structure to store the indexed data.

To drop an index in SQL Server, you can use the `DROP INDEX` command. This command takes the name of the index and the name of the table as arguments, and removes the metadata, statistics, and index pages associated with the index. If you drop a clustered index, the table will become a heap, which means that the physical order of the data in the table will no longer be determined by the index.

It's important to note that once an index has been dropped, it can't be rebuilt. Instead, you need to create a new index with the same name and definition as the original index. This means that if you are going to drop an existing index, you should make sure you have a script of the `CREATE INDEX` statement for the index before you drop it. This way, you can recreate the index later if you need to.

In summary, dropping an index in SQL Server removes the metadata, statistics, and index pages associated with the index, and a dropped index cannot be rebuilt. If you need to drop an index, make sure you have a script of the `CREATE INDEX` statement before you drop it, so you can recreate the index later if necessary.



Cannot be rebuilt, as in the metadata is gone. The index can be stil be created and used for a performance gain, right?



Yes, you are correct. When an index is dropped, the metadata and statistics associated with the index are removed, so the index cannot be rebuilt using the same metadata and statistics. However, you can recreate the index with new metadata and statistics, which will provide the same performance benefits as the original index.

To recreate an index that has been dropped, you need to use the `CREATE INDEX` statement to define the new index, including its name, the columns it will index, and any options or settings for the index. Once the new index is created, it can be used to speed up queries and data retrieval operations.

So to summarize, while a dropped index cannot be rebuilt using the same metadata and statistics, it can be recreated with new metadata and statistics using the `CREATE INDEX` statement. The recreated index will provide the same performance benefits as the original index, and can be used to speed up queries and data retrieval operations.



Tell me one thing, is the index creation process repeatable and deterministic?



Yes, the index creation process is repeatable and deterministic. The `CREATE INDEX` statement defines the index using specific columns and settings, so running the statement again will create the same index with the same definition.

The index creation process is also deterministic in the sense that it will always create the same index with the same properties and settings, given the same input data and index definition. This means that if you create an index on a specific set of columns and with specific settings, the resulting index will be the same every time the `CREATE INDEX` statement is executed on the same data and with the same settings.

However, keep in mind that the performance of an index may depend on factors such as the distribution of data within the table, the amount and type of data being indexed, and the hardware and configuration of the database server. So while the index creation process is repeatable and



deterministic, the performance of the resulting index may vary depending on the specific characteristics of the data and hardware environment.



Case A - Suppose I have some tables, and they have some indexes. I delete all indexes.



If you delete all the indexes on a table, you will remove the metadata associated with those indexes, and the index pages on disk will be deleted. This means that any queries that relied on those indexes will no longer be able to use them to speed up data retrieval.

If you need to recreate the indexes, you will need to use the `CREATE INDEX` statement to define the new indexes, including their names, the columns they will index, and any options or settings for the indexes. Once the new indexes are created, they can be used to speed up queries and data retrieval operations.

However, keep in mind that creating indexes can be a time-consuming process, especially for large or heavily-used databases, so it's important to plan accordingly and allow enough time for the index creation process to complete. Additionally, adding indexes to a table can also affect the performance of insert, update, and delete operations, so it's important to test the impact of the new indexes on the overall performance of the database.



I pressed enter prematurely, let me write again



No problem, take your time.



Case A - Suppose I have some tables, and they have some indexes. I delete all indexes. I recreate the same indexes.

Case B - Suppose I have same tables, on the same hardware machine. I recreate the indexes (that were there initially in case A), it takes some time.

Is the end result same in both cases?



Assuming that the database schema and data are the same in both cases, the end result should be the same after recreating the indexes, regardless of whether the indexes were previously deleted (Case A) or remained in place (Case B).

This is because the `CREATE INDEX` statement is deterministic, meaning that it will create the same index with the same properties and settings every time it is executed on the same set of data and index definition.

However, it's worth noting that recreating indexes can be a time-consuming process, especially for large or heavily-used databases, and can impact the performance of other operations while the indexes are being recreated. It's important to plan accordingly and schedule index creation during off-peak hours or when there is low activity on the database, to minimize any impact on the overall performance of the database.



Let's talk about storing files - text like, or binary ones (images, audio etc) in a database. Let me ask questions



Sure, I'll do my best to answer your questions about storing files in a database. Go ahead!



Files - text like (JSON, .txt, .md) or binary ones (media files) are usually never stored directly in a column of a database. Instead the file is stored in a filesystem (or something of the likes), and a link (URL or filesystem path) is saved to it.

When the database is queried, the link from the column can be used to fetch the file from the filesystem (for local consumption) or the static server (for internet consumption, may even be a CDN).

Reason: This is done because files don't have a homogeneous sizes and therefore the internal data structures of tables will be become more linked-list type (slower access) than array like (fast access).

Verify this situation, and my reason.






Yes, you are correct. In general, it is not recommended to store files directly in a database, whether they are text files or binary files like images or audio. Instead, it is more common to store the files in a file system or a cloud-based storage service, and store a link or reference to the file in the database.

There are a few reasons for this. One reason, as you mentioned, is that files can be quite large and can vary greatly in size, which can make it difficult to store them efficiently in a database. Databases are designed to store structured data in a tabular format, and storing large, unstructured data like

files can cause the internal data structures of the database to become more linked-list-like, which can slow down access times.

Another reason is that storing files in a database can make it more difficult to manage backups and disaster recovery. If the files are stored separately from the database, they can be backed up independently, which can make it easier to recover from data loss or corruption.

By storing a link or reference to the file in the database, you can still associate the file with the relevant data in the database (for example, a user's profile picture or a document associated with a project), while avoiding the performance and management issues that can come with storing large files directly in the database.   



Yes, for example a delete operation for a file can be done in nearly  $O(1)$  time (assuming record can be accessed faster), since the DB actually clears the link to empty and enqueues an async task for deletion of the file from the storage (filesystem, cloud storage), which can be done at any time when the server finds convenient.



Yes, that's correct. By storing a reference or link to the file in the database, it becomes much easier to manage the file's lifecycle, including deleting it when it is no longer needed.

When a file needs to be deleted, the database can simply remove the link or reference to the file from the relevant record or table, which can be done quickly and efficiently. The actual file deletion can then be handled by an asynchronous process or background job, which can be scheduled to run at a convenient time or during periods of low activity on the server.

By separating the storage of the file from the database and using a reference or link instead, you can improve the performance and scalability of your application, while still retaining the ability to easily manage and delete files as needed.





Let's assume, for this discussion that I don't wish to use a dedicated file storage service (like S3). I wish to save the file on the server (not in the DB, but still on the server).

There are two problems to solve here:

1. My DB code (most likely ORM) needs to talk to the filesystem, including async jobs for it (like we discussed for delete)
2. I need to generate a link to be given to the user (client) that obfuscates the file path in the server, but the server can understand the path (if this link is requested).

Are these problems, acknowledged by backend engineers, or are these non-issues?



Yes, these are definitely issues that backend engineers need to consider when implementing a file storage solution that stores files on the server file system instead of using a dedicated file storage service like S3.

To address your first point, where the DB code needs to talk to the file system, you would need to ensure that your application code has the appropriate permissions and access to the file system. This can involve setting up user accounts, permissions, and groups on the server, and configuring your application to use those accounts to access the file system.

In addition, you would need to implement asynchronous jobs or tasks to handle file operations like deletes or updates, as these can be time-consuming and potentially impact the performance of your application if done synchronously. This can involve using a job queue or task runner to handle these tasks in the background, and ensuring that the tasks are resilient and can handle failures or errors gracefully.

For your second point, where you need to generate a link that obfuscates the file path on the server, you would need to implement a routing or URL mapping system that maps the user-facing URLs to the actual file paths on the server. This can involve using a combination of URL rewriting, URL parameters, or query strings to encode the file path in the URL, while still keeping it obfuscated or hidden from the user.

Overall, while it is certainly possible to store files on the server file system instead of using a dedicated file storage service, it does require additional setup and configuration, as well as careful consideration of security, performance, and scalability concerns.





Maybe I can have a table that stores the link <--> filepath, or just add the column directly to the place where link is a column.



Yes, that's definitely an option. You could create a table or add a column to an existing table that maps the link to the corresponding file path on the server file system. This can make it easier to manage the links and file paths in your application code, as you would not need to hard-code the file paths in your code.

If you choose to use a separate table for the link-to-file-path mapping, you could use a primary key on the link column to ensure uniqueness, and add indexes on the link and file path columns to enable fast lookup and querying. You could also add additional columns to the table to store metadata about the file, such as the file type, size, creation date, or owner.

Alternatively, you could add a column directly to the table where the link is stored, to store the corresponding file path on the server file system. This can simplify the design of your database schema, as you would not need to create a separate table for the link-to-file-path mapping. However, it can make it more difficult to manage the file paths if they need to be updated or moved in the future.

Overall, both approaches have their pros and cons, and the choice will depend on your specific requirements and constraints.



And a simple hash function would create the file link, maybe salted with the date. Since our goal here is to uniquely generate a file, and not encrypt anything



Yes, that's definitely an option for generating file links that are unique and not easily guessable. You could use a hash function to generate a unique string based on some input data, such as the file name, file type, or a timestamp, and then append that string to a base URL or file path on the server to create the final link.

To make the hash function more secure and less susceptible to brute-force attacks, you could also add a salt value, which is a random value that is concatenated with the input data before the hash function is applied. This can make it more difficult for attackers to guess the input data and generate valid file links.

Overall, using a hash function with a salt value can be an effective way to generate unique and secure file links, but it does require careful consideration of the hashing algorithm and salt value used, as

well as potential performance and scalability issues if the hash function needs to be applied frequently or for large amounts of data.



I have a question about querying tables. Does a simple 'SELECT someColumnName FROM tableName' perform significantly differently for two tables whose width is different (i.e. one has more columns than the other)?



In general, the number of columns in a table should not have a significant impact on the performance of a simple SELECT query that retrieves only a subset of columns from the table. This is because the database engine can use various optimizations, such as reading only the necessary pages from disk and minimizing the amount of data that needs to be transferred over the network, to retrieve the requested columns efficiently.

However, if the difference in the number of columns is extremely large (e.g. one table has only a few columns, while the other has hundreds or thousands of columns), there may be some performance overhead due to the additional metadata and memory required to store and manage the larger table schema. This can potentially impact the performance of queries that involve joining or filtering on the table, especially if the larger table has many unused or sparsely populated columns that increase the overall size of the table on disk.

In general, it's a good practice to design your database schema to minimize the number of columns in each table, and to use normalization techniques to eliminate redundant data and improve query performance. However, it's also important to balance this with the need to store and manage the data effectively, and to consider the overall performance and scalability of the system.



From an asymptotic POV (which may not be enough for real world apps), there're the same, right - since it's like `arr[i].attributeName`, i.e. everything is done using pointers (in a way)



Yes, from an asymptotic perspective, a simple SELECT query that retrieves a subset of columns from a table should have the same time complexity regardless of the number of columns in the table. This is because the database engine can access the requested columns using pointers or direct memory access, rather than iterating through each column or copying the entire row to a temporary buffer.

However, in practice, the performance of the query can depend on various factors such as the underlying database engine, the size and distribution of the data, the presence of indexes or

constraints, and the load on the system. Additionally, retrieving additional columns that are not needed by the query can increase the overall amount of data that needs to be transferred, leading to slower performance and higher network utilization. Therefore, it's generally a good practice to design tables with only the necessary columns, and to optimize queries to retrieve only the required data.



If this is true, and I don't need indexes, and join. generally. Adding the column instead of creating a separate table for link <--> filepath will be OK, right?



If you don't need to join or query based on the link or filepath column, and you don't expect the table to become too large, then adding the column directly to the table may be a reasonable solution. This can simplify your database schema and make it easier to manage, especially if you don't have any other compelling reasons to separate the link and filepath into a separate table.

However, if you do anticipate that the table may become very large, or if you need to query or join based on the link or filepath column, then it may be better to use a separate table to store the link and filepath information. This can help to keep the main table compact and efficient, while still allowing you to access and manage the link and filepath data in a flexible way. Additionally, a separate table can make it easier to add indexes or other optimizations for querying and joining the data, if needed in the future.



Asymptotically speaking, a simple query (no join or anything) in either case - add column to same table vs create a separate table, will both take the same time right, since  $\text{table1}[i].x$  is  $\log(N)$  and  $\text{table1}[i].x + \text{table2}[j].y$  is also  $\log(N)$



Asymptotically speaking, both options (adding a column to the same table vs creating a separate table) should have similar performance for a simple query that does not involve any joins or complex filtering conditions. In both cases, the query would involve reading a subset of rows from a single table, which can be done efficiently using indexing or other techniques.

However, in practice, there may be some performance differences depending on various factors such as the size and distribution of the data, the underlying database engine, the presence of indexes or constraints, and the specific query being executed. Additionally, there may be other considerations such as data normalization, ease of management, and scalability that may influence your choice of database design.