**Questions about Databases:**

**How would you migrate an application from a database to another, for example from MySQL to PostgreSQL? If you had to manage that project, which issues would you expect to face?**

*My experience with MySQL -> Postgresql migration wasn't really pleasant, so I'd have to second Daniel's suggestion about CSV files.*

*In my case, I recreated the schema by hands and then imported all tables, one-by-one, using mysqldump and pg\_restore.*

*So, while this dump/restore may work for the data, you are most likely out of luck with schema. I haven't tried any commercial solutions, so see what other people say and... good luck!*

*UPDATE: I looked at the code the process left behind and here is how I actually did it.*

*I had a little different schema in my PostgreSQL db, so some tables were joined, some were split. This is why straightforward import was not an option and my case is probably more complex than what you describe and this solution may be an overkill.*

*For each table in PG database I wrote a query that selects the relevant data from MySQL database. In case the table is basically the same in both databases, and there are no joins it can be as simple as this*

*select \* from mysql\_table\_name*

*Then I exported results of this query to XML, to do this you need to run it like this:*

*echo "select \* from mysql\_table\_name" | mysql [CONNECTION PARAMETERS] -X --default-character-set=utf8 > mysql\_table\_name.xml*

*This will create a simple XML file with the following structure:*

*<resultset statement="select \* from mysql\_table\_name">*

*<row>*

*<field name="some\_field">field\_value</field>*

*...*

*</row>*

*...*

*</resultset>*

*Then, I wrote a script, that produces INSERT statement for each row element in this XML file. The name of the table, where to insert the data was given as a command line parameter to this script. Python script, in case you need it.*

*These sql statements were written to a file, and then fed to psql like this:*

*psql [CONNECTION PARAMETERS] -f FILENAME -1*

*The only trick there was in XML -> SQL transformation is to recognize numbers, and unquote them.*

*To sum it up: mysql can produce query results as XML and you can use it.*

[*https://stackoverflow.com/questions/4756825/mysql-to-postgresql-migration*](https://stackoverflow.com/questions/4756825/mysql-to-postgresql-migration)

**Why databases treat null as a so special case? For example, why in SQL SELECT \* FROM table WHERE field = null does not match records with null field?**

*SQL NULL's special, and you have to do WHERE field IS NULL, as NULL cannot be equal to anything,*

**ACID is an acronym that refers to Atomicity, Consistency, Isolation and Durability, 4 properties guaranteed by a database transaction in most of the database engines. What do you know about this topic? Would you like to elaborate?**

*261*

*ACID is a set of properties that you would like to apply when modifying a database.*

*Atomicity*

*Consistency*

*Isolation*

*Durability*

*A transaction is a set of related changes which is used to achieve some of the ACID properties. Transactions are tools to achieve the ACID properties.*

*Atomicity means that you can guarantee that all of a transaction happens, or none of it does; you can do complex operations as one single unit, all or nothing, and a crash, power failure, error, or anything else won't allow you to be in a state in which only some of the related changes have happened.*

*Consistency means that you guarantee that your data will be consistent; none of the constraints you have on related data will ever be violated.*

*Isolation means that one transaction cannot read data from another transaction that is not yet completed. If two transactions are executing concurrently, each one will see the world as if they were executing sequentially, and if one needs to read data that is written by another, it will have to wait until the other is finished.*

*Durability means that once a transaction is complete, it is guaranteed that all of the changes have been recorded to a durable medium (such as a hard disk), and the fact that the transaction has been completed is likewise recorded.*

*So, transactions are a mechanism for guaranteeing these properties; they are a way of grouping related actions together such that as a whole, a group of operations can be atomic, produce consistent results, be isolated from other operations, and be durably recorded.*

[**https://stackoverflow.com/questions/3740280/acid-and-database-transactions**](https://stackoverflow.com/questions/3740280/acid-and-database-transactions)

**How would you manage database schema migrations, that is, how would you automate the changes a database schema is affected to, as the application evolve, version after version?**

*I have a "Schema" object that I use - but you could do the same without classes..*

*What you want to do is create a 'db\_schema\_versions' table:*

*CREATE TABLE db\_schema\_versions (*

*`table` varchar(255) NOT NULL PRIMARY KEY,*

*`version` INT NOT NULL*

*)*

*After your database can track what version # it is on - it can do SQL upgrades automatically.*

*You should lock your schema table while upgrading schema. This way you wont have two requests at the same moment trying to upgrade your schema.*

*So - keep track of the version you are upgrading from - build a big switch - something like this:*

*class SNTrack\_Db\_Schema extends MW\_Db\_Schema\_Abstract {*

*protected $table = "sntrack\_db\_schema";*

*protected $version = 5;*

*protected function upgrade($fromVersion) {*

*// don't break*

*switch($fromVersion) {*

*case 0:*

*$this->db->query('CREATE TABLE sntrack\_inbound\_shipment (*

*`id` INT NOT NULL PRIMARY KEY AUTO\_INCREMENT,*

*`from` VARCHAR(255) NOT NULL,*

*`date` DATE NOT NULL,*

*`invoice` VARCHAR(255) NOT NULL,*

*`notes` TEXT*

*)');*

*$this->setVersion(1);*

*case 1:*

*$this->db->query('ALTER TABLE sntrack\_details ADD `shipment\_id` INT');*

*$this->db->query('ALTER TABLE sntrack\_product ADD `inventory` INT NOT NULL DEFAULT 0');*

*$this->db->query('CREATE TABLE sntrack\_inventory\_shipment (*

*`shipment\_id` INT NOT NULL,*

*`product\_id` INT NOT NULL,*

*`qty` INT NOT NULL,*

*PRIMARY KEY (`shipment\_id`, `product\_id`)*

*)');*

*$this->setVersion(2);*

*...etc*

[*https://stackoverflow.com/questions/825787/how-to-automate-migration-schema-and-data-for-php-mysql-application*](https://stackoverflow.com/questions/825787/how-to-automate-migration-schema-and-data-for-php-mysql-application)

**How is Lazy Loading achieved? When is it useful? What are its pitfalls?**

*Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used. The opposite of lazy loading is eager loading.*

*It's called lazy loading because, like a lazy person, you are putting off doing something you don't want to. The opposite is Eager Loading, where you load something right away, long before you need it.*

*If you are curious why people might use lazy loading, consider an application that takes a LOOOOONG time to start. This application is probably doing a lot of eager loading... loading things from disk, and doing calculations and whatnot long before it is ever needed.*

*Compare this to lazy loading, the application would start much faster, but then the first time you need to do something that requires some long running load, there may be a slight pause while it is loaded for the first time. Thus, with lazy loading, you are amortizing the load time throughout the course of running your application... and you may actually save from loading things that the user may never intend to use.*

**The so called "N + 1 problem" is an issue that occurs when the code needs to load the children of a parent-child relationship with a ORMs that have lazy-loading enabled, and that therefore issue a query for the parent record, and then one query for each child record. How to fix it?**

*The N + 1 problem is a performance anti-pattern in which an application makes N + 1 database calls (where N is the number of objects fetched). Like most antipatterns, this isn’t necessarily a problem in itself, but under certain circumstances (where N is large, for example) it will cause performance to degrade by making hundreds or even thousands of database calls for a single business transaction.*

[*https://blog.appdynamics.com/product/common-application-problems-and-how-to-fix-them-the-select-n-1-problem/*](https://blog.appdynamics.com/product/common-application-problems-and-how-to-fix-them-the-select-n-1-problem/)

**How would you find the most expensive queries in an application?**

**In your opinion, is it always needed to use database normalization? When is it advisable to use denormalized databases?**

*Denormalization is a time-space trade-off. Normalized data takes less space, but may require join to construct the desired result set, hence more time. If it's denormalized, data are replicated in several places. It then takes more space, but the desired view of the data is readily available.*

*There are other time-space optimizations, such as*

*denormalized view*

*precomputed columns*

*As with any of such approach, this improves reading data (because they are readily available), but updating data becomes more costly (because you need to update the replicated or precomputed data).*

*Pros and Cons of a Normalized database design.*

*Normalized databases fair very well under conditions where the applications are write-intensive and the write-load is more than the read-load. This is because of the following reasons:*

*Normalized tables are usually smaller and have a smaller foot-print because the data is divided vertically among many tables. This allows them to perform better as they are small enough to get fit into the buffer.*

*The updates are very fast because the data to be updated is located at a single place and there are no duplicates.*

*Similarly the inserts are very fast because the data has to be inserted at a single place and does not have to be duplicated.*

*The selects are fast in cases where data has to be fetched from a single table, because normally normalized tables are small enough to get fit into the buffer.*

*Because the data is not duplicated so there is less need for heavy duty group by or distinct queries.*

*Although there seems to be much in favor of normalized tables, with all the pros outlined above, but the main cause of concern with fully normalized tables is that normalized data means joins between tables. And this joining means that read operations have to suffer because indexing strategies do not go well with table joins.*

*Now lets have a look at the pros and cons of a denormalized database design.*

*Pros and cons of denormalized database design.*

*Denormalized databases fair well under heavy read-load and when the application is read intensive. This is because of the following reasons:*

*The data is present in the same table so there is no need for any joins, hence the selects are very fast.*

*A single table with all the required data allows much more efficient index usage. If the columns are indexed properly, then results can be filtered and sorted by utilizing the same index. While in the case of a normalized table, since the data would be spread out in different tables, this would not be possible.*

*Although for reasons mentioned above selects can be very fast on denormalized tables, but because the data is duplicated, the updates and inserts become complex and costly.*

*Having said that neither one of the approach can be entirely neglected, because a real world application is going to have both read-loads and write-loads. Hence the correct way would be to utilize both the normalized and denormalized approaches depending on situations.*

*http://www.ovaistariq.net/199/databases-normalization-or-denormalization-which-is-the-better-technique/#.XDsfS1Uzbec*

**Of of the Continuous Integration's techniques is called Blue-Green Deployment: it consists in having two production environments, as identical as possible, and in performing the deployment in one of them while the other one is still operating, and than in safely switching the traffic to the second one after some convenient testing. This technique becomes more complicated when the deployment includes changes to the database structure or content. I'd like to discuss this topic with you.**

*Why and How Database Changes Should Be Included in the Deployment Pipeline*

*Databases, although different from applications, can and should be included in the same development process as applications. We call this database shift left.*

*When all database changes are described with scripts, source control can be applied and the database development process can include continuous integration and continuous delivery activities, namely taking part in the deployment pipeline.*

*Automation is the special ingredient that powers up the deployment pipeline making it repeatable, reliable and fast, reducing fear for database changes.*

*Migrations-based and state-based are two different approaches to describing a database change. Independently of the choice, small batches are always a good choice.*

*The deployment pipeline is a technical and cultural tool where DevOps values and practices should be reflected according to the needs of each organization.*

*https://www.infoq.com/articles/deployment-pipeline-database-changes*