Database Basics: SQL, MySQL Configuration

These days an SA must know something about databases. A *database* is just a collection of data. Usually this collection is highly structured into *records*. Even a simple file containing structured data, such as the `/etc/passwd` file, may be considered a database. (The SQL standard calls databases *catalogs*, but the two terms are the same in practice.)

Most databases must retain the data for longer than a single user session. This means the data must be saved using **persistent storage**, a major part of nearly all computer software including system software, applications, and web (and other network) services. (Persistent storage technology today generally means hard disks.)

Create, read, update, and *delete* (***CRUD***) are the four basic functions of persistent storage, a major part of nearly all computer software including many web applications. CRUD refers to all of the major functions that need to be implemented in a relational database application or RESTful (Representational State Transfer) web application to consider it complete.

Databases can be classified in various ways. One way is to consider the main purpose of the application (i.e., the most common use of the database): ***[On-Line] Transaction Processing databases*** (OLTP) are optimized for the CRUD operations used to capture data, whereas ***decision-support databases*** (or OLAP, *on-line analytical processing*) are optimized for query operations used to analyze the data.

Data for decision-support systems is often captured by online transaction-processing systems, extracted and transformed to a form suitable for analysis, and then loaded into a decision-support system (i.e., a separate DB). This process is called ETL (*extract, transform, load*). The resulting database is called a ***data warehouse***, giving us the term *data warehousing*.

> ***[From: publib.boulder.ibm.com]***
> **Transaction-processing systems** (OLTP) are designed to capture information and to be updated quickly. They are constantly changing and are often online 24 hours a day. Examples of transaction-processing systems include order entry systems, scanner-based point-of-sale registers, automatic teller machines, and airline reservation applications. These systems provide operational support to a business and are used to run a business.
>
> ***Decision-support systems*** (OLAP, although no one knows what the *on-line* part is supposed to mean here) are designed to allow analysts to extract information quickly and easily. The data being analyzed is often

historical: daily, weekly, and yearly results. Examples of decision-support systems include applications for analysis of sales revenue, marketing information, insurance claims, and catalog sales. A decision-support database within a single business can include data from beginning to end: from receipt of raw material at the manufacturing site, entering orders, tracking invoices, and monitoring database inventory to final consumer purchase. These systems are used to manage a business. They provide the information needed for business analysis and planning.

SAs must be able to setup and manage DBs for developers, testers, and maintainers of applications, for IT administrative uses (such as single sign-on, IP address maps, asset management, trouble ticketing, wikis, CMS, etc.), for web sites, and for business management use , e.g. CRM (customer relationship management systems such as SAP and SalesForce.com) or ERP (enterprise resource planning). Many of these types of DBs require periodic tasks or other maintenance.

The SA must also setup filesystems and storage volumes to hold DB data (not always kept in files) and to set appropriate mount and I/O options. This must be done by working with a DBA and/or developer, or the DB performance is likely to be very bad.

Below, we discuss the commonly used ***relational database***. *Relational* is a mathematical term that simply means based on tables. Briefly, a database consists of *tables* of data, with each *row* representing data related to a single entity (that is, each row is a record) and each *column* an attribute. This notion is very powerful, and allows data to be searched quickly, to answer various *queries*. Most applications use such relational databases, as these support commonly needed business functions.

Non-Relational Databases

Not all databases are relational. For IT purposes an object orientated, **hierarchical database** is often used, usually via an LDAP server. These are often called *directories* and not databases. An example of such a DB is the global DNS system.

More recently, so-call "NoSQL" (non-relational) databases have become popular for web services and other uses. These are generally variations of key (name)—value stores. CouchDB is a *document-oriented* database that stores structured JSON blobs with nested key/value pairs. It is designed primarily to store configuration data (like `dconf`). CouchDB has a built-in Web server that is used by applications to communicate with the database. Other popular non-relational databases include MongoDB and Cassandra. Another NoSQL DB, designed for

social networking sites, is called Stig. Amazon has also opened the NoSQL DB it has used internally for years, as a service called DynamoDB.

> The most widely used key-value database must be the **Berkley Database** (BDB). Acquired by Oracle in 2006, BDB is still actively being developed. BDB supports multiple data items for a single key, can support thousands of simultaneous threads of control or concurrent processes, and can manipulate databases as large as 256 terabytes, on a wide variety of operating systems.

When you manage clusters of servers or containers, all must share some data, especially configuration data. This is often done using one of several key-value databases designed for such purposes (e.g., highly reliable and replicated). Examples include Apache Zookeeper, CoreOS etcd (see also Github), HashiCorp Consul. Some of these tools provide additional functionality needed for clusters of VMs or containers, such as message brokering or service discovery.

(In-memory databases are popular too, such as redis and memcached.)

> The most widely used key-value database must be the **Berkley Database** (BDB). Acquired by Oracle in 2006, BDB is still actively being developed. BDB supports multiple data items for a single key, can support thousands of simultaneous threads of control or concurrent processes, and can manipulate databases as large as 256 terabytes, on a wide variety of operating systems.

When you manage clusters of servers or containers, all must share some data, especially configuration data. This is often done using one of several key-value databases designed for such purposes (e.g., highly reliable and replicated). Examples include Apache Zookeeper, CoreOS etcd (see also Github), HashiCorp Consul. (In-memory databases are popular too, such as redis and memcached.) Some of these tools provide additional functionality needed for clusters of VMs or containers, such as message brokering or service discovery.

another type of database is called a ***time-series*** database, used for storing event and metric data. Such databases are commonly used for monitoring systems.

> ***Blockchain*** is a new (2008) type of database, used for many purposes. A blockchain serves as the basis for some cryptocurrencies such as Bitcoin (the first public blockchain) and Ether. At its heart, a blockchain is a list of blocks of records. Every so often, a group of records is collected into a block which is added at the end of the list. Strong cryptographic methods are used making any changes to the blockchain very difficult. This means the blockchain can be used as a ledger that cannot be rewritten and can be verified by anyone with access. A typical blockchain block holds

transaction records (each digitally signed), plus a hash of the previous block, a timestamp, and possibly other data. As of 2017, many companies are experimenting with blockchain technology to see if it can work better than traditional databases for particular applications.

Relational Databases

As mentioned above, in a relational database each *row* of a table is considered a *record* that contains data related to some object or entity: a person, a product, an event, an order, etc. The rows contain columns called *attributes* or fields, each with a name, a type, and possibly some constraints. A `Person` table might look like this:

ID Number	Name	Title	Phone
0001	John Public	Anyman	555-1234
0002	Jane Doe	President	555-4321

Given such a table, you can ask queries such as "*what is the name of person 0002?*" and "*what is the phone number of Jane Doe?*".

Schemas

The design for a database affects its usability and performance in many ways, so it is important to make the initial investment in time and research to design a database that meets the needs of its users. **A database schema is the design or plan of the database** including:

- what data is to be stored
- the type of each piece of data (e.g., text, number, phone number, date, ...)
- how the data will be organized (i.e., what tables to have)
- any constraints or limits on the data (e.g., number ranges, length of text string)
- how the various data relates to one another.

The term *schema* has another meaning: a group of database objects (that is, tables, views, indexes, stored procedures, triggers, sequences, etc.). In this sense, a schema is a ***namespace***, used to assign conveniently permissions to a number of objects (tables), and to allow reuse of (table) definitions in several different databases. Usually there is a default schema for a given database.

A user can access objects in any of the schemas in any database they can connect to, provided they have the proper privileges.

A ***database management system*** (DBMS) handles all the actual file reading, writing, locking, flushing, and in general handles all the details of the CRUD operations so the data is efficiently and safely managed. It also handles other common operations such as managing network parameters, database creation, schema definition, security, etc., that are needed to work with databases.

Once set up, a DBMS system can be used by an application to read data, parse it, and store it, so it can be efficiently searched and retrieved later. An application *connects* to the DBMS, indicates which database to use, and supplies a username and password.

> A given DBMS many run several independent ***instances*** on a given server. Each instance may manage one or more *databases* (catalogs), which contain the tables from one or more *schemas*.

Once connected, an application sends various query and update (CRUD) statements to the DBMS. Note that these query and update statements (often written using a standard language such as SQL) only say *what* you want (*declarative programming*). Thus SQL differs from most programming languages in which you must express *how* to do something (*imperative programming*).

The database can be structured in various ways: plain old files, in tables of rows and columns, or as named objects organized in a hierarchy. So why use a DBMS?

Enterprise applications all have similar data storage needs: they often require concurrent access to distributed data shared amongst multiple components, and to perform operations on data. These applications must preserve the *integrity* of data (as defined by the business rules of the application) under the following circumstances:

- distributed access to a single resource of data, and
- access to distributed resources from a single application component.

Plain files don't support this use. In the old days (1950s–1970s) programmers decided what questions were going to be asked (for decision support DBs) or what data to capture (for OLTP DBs), designed a ***schema*** for the data (what tables and columns were needed), and implemented the whole thing in COBOL (shudder). But soon it was realized that these enterprise applications all had similar needs and differed only in the specific schema. It was a waste of time to re-implement the same functionality afresh in each application. Putting the common parts in a DBMS greatly speeds database application development and helps ensures the functionality is well-implemented and error free.

> Note that while a single DBMS can serve multiple databases simultaneously, in practice the network bandwidth requirements, large disk space requirements, and different security and backup policies make this

impractical. Having one host running one DBMS which serves a single database is a common practice.

A ***Relational Database Management System*** (**RDBMS**) is a system that allows one to define multiple databases simply by providing the schemas, and can preserve the data integrity. Today's RDBMSes do this very well; some can support a huge number of tables, with a huge number of rows of data (terabytes and more), for hundreds of simultaneous clients. Most support additional features and management tools as well.

All RDBMSes today support a common language used to define schemas and queries: ***SQL*** or ***Structured Query Language***. The language has three parts, the ***Data Definition Language*** (DDL, the SQL where you define and change schemas) and the ***Data Manipulation Language*** (DML, the SQL where you lookup, add, change, or remove data). The third part is used to manage the server and the databases; this may be called ***Data Control Language*** or DCL. However this is the most recently standardized part of SQL and the least well supported; most DBMSs use non-standard commands for this.

SQL supports software's need for CRUD. Each letter in the acronym *CRUD* can be mapped to a standard SQL statement: INSERT, SELECT, UPDATE, and DELETE.

System administrators need to be most familiar with DDL and DCL, since it will usually be their job to manage the DBMS and to create and manage the databases. Software developers need to be most familiar with DDL and DML. A DBA should be expert with all parts. But everyone should know something about each part of SQL.

Although SQL is an ISO and ANSI standard, most RDBMSes only partially support the standard or add proprietary extensions that are very useful. This makes changing your RDBMS vendor difficult, as migrating your data, schemas, and queries can be painful. It doesn't help that the standard changes dramatically every four years or so, and that some parts of the standard are marked as optional. Here is a brief list of the SQL standard versions (From Wikipedia):

- 1986 — First adopted by ANSI and commonly called "SQL-86" or "SQL-87".
- 1992 — Major revision adding many missing features, and adopted as ISO 9075, commonly called "SQL2", "SQL92" or sometimes "ANSI-SQL" (all versions are standardized by ANSI).
- 1999 — Major revision adding new data types, regular expressions, some procedural statements, triggers, and more; commonly called "SQL3", "SQL99", or "SQL:1999".

- 2003 — Added some XML support, sequences, and automatically generated column values (e.g., the next ID number); commonly called "SQL:2003".
- 2006 — Added significant XML support.
- 2008 — Minor revision but with some new features added.
- 2011 — Minor revision with some new features added (notably, support for temporal tables, where each row is time-stamped and you can search for data within some period of time).
- 2016 — The current version (as of 2018). Added some major new features, including JSON support, Regular Expression matching of rows, and date/time formatting and parsing.

In addition to supporting different sub-sets of SQL (all modern ones support at least SQL-92), different RDBMSes support different configuration methods and security models, and need expertise for *tuning* the system (adjusting RDBMS parameters and re-working some queries and schemas) to provide good performance.

> A modern DBMS reads in the query and generates several possible ***execution plans***. Each plan is essentially a program; a series of low-level disk access operations. All of the plans are correct; when run, each plan results in the same answer to the query. They differ only in their efficiency. Picking the wrong plan can make a large difference in the time it takes when answering the query. The various execution plans are compared using cost-based query optimizers, and the most efficient (lowest cost) one is chosen and used.
>
> A badly tuned system can take hours/days rather than seconds/minutes for some operations! It is up to a database administrator ("DBA") to tune the DBMS by setting various parameters, so it stores the data efficiently and generates efficient execution plans.

Comparing Popular RDBMSes

By far the most capable and popular commercial RDBMS is **Oracle** (about 40% market share), with IBM's **DB2** also popular (~33%). MS SQL server has about 11% (as reported by IDC at databases.about.com '10). However in recent years a number of open source alternatives have established themselves: **MySQL** and **PostgreSQL** (Postgres) are two common ones (with a reported market share second only to Oracle). There are free versions of all popular RDBMSes available.

Of the open source DBMSes, MySQL is (currently, 2010) more popular than PostgreSQL. It is very fast for certain applications, and works very well with

PHP, so has become a de facto standard for web development (***LAMP***: Linux, Apache, MySQL, and PHP). The heart of any DBMS is the DB engine. MySQL supports several, each tuned for a different purpose. The MyISAM engine is the fast one, but it isn't suitable for OLTP. The InnoDB supports features similar to PostgreSQL and other RDBMSes, but is not very fast. InnoDB was bought by Oracle (while the rest of MySQL was bought by Sun). FYI: The DB engine for Microsoft Access DBMS is called the "Jet" engine.)

> In 2010, Oracle bought Sun Microsystems and now owns all their assets, including MySQL. In 2011, Oracle added some proprietary enhancements to MySQL, moving it toward a non-free, non-open source model; its future is uncertain. In 2013, a number of systems including Fedora have announced a switch to MariaDB.
>
> MariaDB is a community-developed "fork" of MySQL, released under the GPL. Its lead developer is Monty Widenius, the founder of MySQL, who named both products after his daughters My (?) and Maria.
>
> Another fork of MySQL worth knowing is Percona, developed by the former performance engineer of MySQL.

PostgreSQL supports more of the current SQL standard and has advanced features (e.g., multiple schemas per DB), and can be very fast for some uses. It's a fine all-around RDBMS. It is also becoming popular as developers shy away from the uncertain future of MySQL. It is highly recommended for new deployments when you don't have a legacy MySQL system to worry about.

Small DB libraries to embed in your application such as SQLite are popular too. (These support one client with one DB, but that is fairly common.) For more insight on the differences between popular RDBMSes see at `wikipedia.org`, "Comparison_of_relational_database_management_systems".

> Derby, H2 (which is currently (2020) popular with Java developers), and HSQLDB are good for small to medium databases (up to a few tens of millions of rows each for dozens of tables). SQLite is good for small databases but only embedded ones; it lacks the features of the others.
>
> MySQL (now owned by Oracle) its successor, MariaDB, and PostgreSQL are good for Gigabyte sized databases that require fast connections such as for websites. (PostgreSQL is likely overall better but MariaDB is optimized for fast reads.)
>
> Go with Oracle or DB2 for Terabyte sized DBs. (Some DBs such as Google's main database are measured in Petabytes!)
>
> Note, even a small DB needs to be well designed (including proper indexes) or it will suffer serious performance issues.

Derby, H2, and some others support **in-memory databases**, very useful when learning and during development! An in-memory DB is especially useful for testing.

Defining Relational Databases

When defining a relational database, you need to specify the database name, how (and by whom) it can be accessed, and the schema that defines the various tables in the database. (Other items may be defined as well, such as procedural functions, triggers, sequences, views, etc.) The heart of the database is the schema; for each table, you need to specify the name of the table, the attributes' names and their datatypes, and any constraints on the columns or the table as a whole. Tables can be defined with the SQL `CREATE TABLE` statement. After the table is created, the schema can be changed with the `ALTER TABLE` statement, but this can be dangerous and slow if the table already contains lots of data.

Datatypes are the names given to the types of each attribute (column), but are not well standardized. Common ones include Boolean, integer, float, fixed-length and variable-length strings, binary objects, as well as currency, dates, times, and intervals. Every attribute must be assigned a datatype.

Constraints are used to limit the type of data that can go into a table. Some of the commonly available constraints (depends on the DBMS used) are:

- **NOT NULL** A *null* value is a special value that means "no data". Use this constraint to prevent creation of a row of data with a null for some column.
- **UNIQUE** Unique means no duplicates allowed in that column.
- **PRIMARY KEY** A *primary key* is a column (or set of columns) that are not null and unique. A table can only have one primary key defined. Usually an index is created automatically for the primary key.
- **FOREIGN KEY** A *foreign key* constrains the values in a column to be primary keys from some other (specified) table. This is useful to prevent updates from causing inconsistent data. This is sometimes called ***referential integrity***. The techniques known as *cascading update* and *cascading delete* ensure that changes made to the linked table are reflected in this table.
- **CHECK** This constraint is used to limit the value range that can be placed in a column. A CHECK constraint on a single column it allows only certain values for this column; on a table, it can limit the values in certain columns based on values in other columns in the row. The constraint is some Boolean expression, such as "`age > 0`".

- **DEFAULT** Not really a constraint, it specifies a default value to be inserted for new rows that don't specify a value for that column. (Has no effect on pre-existing rows.)

Joins and Referential Integrity

Foreign key constraints depend on data from multiple tables. The data in multiple tables are linked using an operation called a "join". A ***join*** essentially builds a composite table from two (or more) tables that have a common column. For example, suppose you have a *book* table with a *book_number*, *title*, and *publisher_code*, and a *publisher* table with *publisher_code* and *publisher_name*. Then you can do a `SELECT` (or other action) on the composite of these to show the *title* and *publisher_name*. You can also ensure only valid *publisher_code* values are added to the *book* table.

There are four types of joins, but it is probably enough for a system administrator to just know the names of them: an *inner join* (the column has the same value in both tables), a *left (outer) join* (all the rows from the left table even if no matching value in the right table), a *right (outer) join*, and a *full (outer) join* (all rows from both tables become rows in the composite table). With outer joins, missing values show as nulls. Also note that nulls don't match anything, not even other nulls, and should be prevented by the schema when possible.

Normalization

Normal forms are a way to prevent DML operations from destroying real data or creating false data. That can happen if the schema isn't designed for the types of queries and multi-user activity that is common. An un-normalized schema is considered to be in normal form 0 (zero).

If a database is not normalized, many sorts of errors are possible and it becomes the application programmers' job to prevent that. A large enterprise application may access a DB from many different places, and all of them need to be carefully written and updated when the database does. On the other hand, if a relational database is normalized, most such errors are intrinsically impossible. Normalization is usually a job for a database administrator (or a knowledgeable lead programmer). Even if that isn't you, you will need to communicate about the schema with such people, so system admins need to know about this.

(Note, OLAP databases are generally not normalized and often have tables with hundreds or thousands of columns. OLTP databases should be normalized.)

Modifying schemas into normal forms is a straight-forward process of transforming a schema from normal form *n* to normal form *n+1*. Although there are many normal forms (at least 9, or over 300, depending on how you count them), normal forms 4 and higher cover obscure potential problems that very

rarely ever manifest, or can be dealt with in other ways. Practically, most DB schema designers are happy with third normal form.

[The following example was adapted from *Joe Celko's SQL for Smarties*, 2nd Ed. (C)2000 by Morgan Kaufmann Pub., chapter 2.]

Consider a schema for student course schedules. The original design might be something like this:

```
Classes (name, secnum, room & time,
  max seats available, professor's name,
  list of students (1..max seats available) )
```

where each `student` has (`name`, `major`, `grade`).

First Normal Form requires no repeating groups; each column value must be a single value and not a list as stated above. The `Classes` schema violates this by having an attribute "list of students". This schema can be converted (*normalized*) to 1st NF as a single SQL table, where each row can be uniquely identified by the combination of (`course`, `secnum`, `studentname`); that becomes the composite primary key. The SQL for the revised schema would be something like this:

```
CREATE TABLE Classes
(course CHAR(7) NOT NULL,
 secnum INTEGER NOT NULL,
 time INTEGER NOT NULL,
 room CHAR(7) NOT NULL,
 maxSeatsAvail INTEGER NOT NULL,
 profname CHAR(25) NOT NULL,
 studentname CHAR(25) NOT NULL,
 major CHAR(15) NOT NULL,
 grade CHAR(1),
 PRIMARY KEY (course, secnum, studentname)
);
```

This schema is in first normal form, but still leads to various ***anomalies***:

- If Prof. Pollock wins lotto and quits, you could delete all his classes by deleting all rows with `profname="Pollock"`. But this also deletes the information about what students are taking Unix/Linux classes (*deletion anomaly*).
- If a student changes a computer course to, say an English poetry course, the database will suddenly show Prof. Pollock as teaching poetry (*update anomaly*).

- If HCC hires a new instructor, there is no way to store that until that instructor has been assigned at least one class with at least one student in it (*insertion anomaly*).

With this schema, it would be up to the application code to also update the other attributes in the row where a student changed their course. This is difficult and error-prone to fix in all the application code that uses some DB, requiring complex application update logic and query checking. Even with all that, not all these problems go away.

Such ad-hoc solutions are impossible to maintain over the long run as your database grows. Many of these problems fade away if each table represents a single fact only. That means the queries may work on several tables at once, but a RDBMS is designed for exactly that.

Second Normal Form breaks up tables from a schema in 1st NF that represent more than one fact into multiple tables, each representing a single fact. This can be understood with the idea of a *table key*. Each table should have a column or group of columns that uniquely identifies a given row. In the schema above the key is (`course`, `secnum`, `studentname`). In 2nd NF, no subset of a table key should be useable to uniquely identify any non-key columns in a table. If they can then the table represents multiple facts.

Our table violates 2nd NF since (`student`, `course`) alone determine the (`secnum`) (and thus all other columns). Checking for other column dependencies shows (`studentname`) determines (`major`).

To transform this 1st NF schema into a 2nd NF one, we need to make sure that every column of each table depends on the entire key for that table. Apparently our database represents three "facts": data about courses, data about sections, and data about students. One possible way to convert the schema into 2nd NF is to split the one table into three tables like this (note the additional constraints used, just to show how to use them):

```
CREATE TABLE Classes
(course CHAR(7) NOT NULL,
 secnum INTEGER NOT NULL,
 time INTEGER NOT NULL,
 room CHAR(7) NOT NULL,
 maxSeatsAvail INTEGER NOT NULL,
 profname CHAR(25) NOT NULL,
 PRIMARY KEY (course, secnum),
 FOREIGN KEY(secnum) REFERENCES Sections(secnum)
);
```

```
CREATE TABLE Students
(studentname CHAR(25) NOT NULL,
 major CHAR(15) NOT NULL,
 PRIMARY KEY (studentname)
);

CREATE TABLE Sections
(secnum INTEGER NOT NULL,
 studentname CHAR(25) NOT NULL,
 grade CHAR(1),
 PRIMARY KEY (secnum, studentname),
 FOREIGN KEY(studentname) REFERENCES
      Students(studentname),
 CHECK (grade IN ("A", "B", "C", "D", "F", "I"))
);
```

However, this schema is also not in second normal form! The Sections table represents information about both sections and about student grades. After splitting that table into two, the final 2nd NF schema becomes:

```
CREATE TABLE Classes
(course CHAR(7) NOT NULL,
 secnum INTEGER NOT NULL,
 profname CHAR(25) NOT NULL,
 PRIMARY KEY (course, secnum),
 FOREIGN KEY(secnum) REFERENCES Sections(secnum)
);

CREATE TABLE Students
(studentname CHAR(25) NOT NULL,
 major CHAR(15) NOT NULL,
 PRIMARY KEY (studentname)
);

CREATE TABLE Sections
(secnum INTEGER NOT NULL,
 time INTEGER NOT NULL,
 room CHAR(7) NOT NULL,
 maxSeatsAvail INTEGER NOT NULL,
 PRIMARY KEY (secnum),
);

CREATE TABLE StudentGrades
(secnum INTEGER NOT NULL,
 studentname CHAR(25) NOT NULL,
 grade CHAR(1),
```

```
 PRIMARY KEY (secnum, studentname),
 FOREIGN KEY(studentname) REFERENCES
      Students(studentname),
 FOREIGN KEY(secnum) REFERENCES Sections(secnum),
 CHECK (grade IN ("A", "B", "C", "D", "F", "I"))
);
```

This four table schema can answer the same queries as the original single table one, but those queries and updates will be more complex. For example, to answer the question *what courses is a given student taking?* or *what is the grade for a given student in a given course?*, you will need to use queries with joins.

> If you're wondering why the primary key for table `Classes` is not just `secnum`, it's because at my school the section numbers can be reused. The real key is probably `(secnum, year, term)`, but I didn't wish to clutter up the example with all the attributes that would be required in the "real-world".

[*Skip rest of normalization except if high interest, and time is available.*]

Although many anomalies are now addressed, notice that `maxSeatsAvail` not only depends on the key for `Sections`, but also on the `room` column. This is sometimes called a *transitive dependency*: `room` depends on `section` and `maxSeatsAvail` depends on `room`. Such a dependency is only acceptable in certain cases, and those require careful application logic so the data doesn't get corrupted. This leads to...

Third Normal Form transforms a schema in 2nd NF by splitting up tables even more than was needed for 2nd NF. To split up the table to remove the transitive dependency, note that 2nd (and 3rd) NF might have multiple possible keys for a table. One is the ***primary key*** and the others are called ***candidate keys***. This notion of candidate keys can be used to define 3rd NF:

> **In 3rd NF, suppose X and Y are two columns of a table. If X implies (determines) Y, then either X must be the (whole) primary key, or Y must be (part of) a candidate key.**

`maxSeatsAvail` has this problem: `room` is not the primary key nor part of any candidate key, but `maxSeatsAvail` depends (only) on `room`. To transform this schema into 3rd NF we split the `Sections` table into `Sections` and `Rooms` tables:

```
CREATE TABLE Classes
(course CHAR(7) NOT NULL,
 secnum INTEGER NOT NULL,
 profname CHAR(25) NOT NULL,
```

```
 PRIMARY KEY (course, secnum),
 FOREIGN KEY(secnum) REFERENCES Sections(secnum)
);

CREATE TABLE Students
(studentname CHAR(25) NOT NULL,
 major CHAR(15) NOT NULL,
 PRIMARY KEY (studentname)
);

CREATE TABLE Sections
(secnum INTEGER NOT NULL,
 time INTEGER NOT NULL,
 room CHAR(7) NOT NULL,
 PRIMARY KEY (secnum)
);

CREATE TABLE Rooms
(room CHAR(7) NOT NULL,
 maxSeatsAvail INTEGER NOT NULL,
 PRIMARY KEY (room)
);

CREATE TABLE StudentGrades
(secnum INTEGER NOT NULL,
 studentname CHAR(25) NOT NULL,
 grade CHAR(1),
 PRIMARY KEY (secnum, studentname),
 FOREIGN KEY(studentname) REFERENCES
      Students(studentname),
 FOREIGN KEY(secnum) REFERENCES Sections(sec-
num),
 CHECK (grade IN ("A", "B", "C", "D", "F", "I"))
);
```

Any good database book (see reviews at www.ocelot.ca/design.htm) will show you how to address other problems with additional normal forms. For example, this schema still allows multiple sections to be assigned the same room at the same time, or one professor teaching multiple courses at the same time. A good schema would make (most) such anomalies impossible. The alternative is to design queries, inserts, updates, and deletions very carefully, with extra care taken to locking tables (to prevent data corruption from simultaneous queries and updates). Obviously it is better if the schema design prevents such corruption from ever occurring.

It isn't a system administrator's job to create schemas for most of the organization's databases. But SAs are expected to be able to create simple schemas for IT uses, and to understand normal forms in general in order to work with DBAs and developers.

Transactions

It is often required that a group of operations on (distributed) resources be treated as one unit of work. In a unit of work, all the participating operations should either succeed or fail (and recover) together. In case of a failure, all the resources should bring back the state of the data to the previous state (i.e., the state prior to the commencement of the unit of work). (Ex: transfer money between accounts.)

The concept of a ***transaction***, and a transaction manager (or a transaction processing service) simplifies construction of such enterprise level distributed applications while maintaining integrity of data. A transaction is a unit of work that has the following properties:

- **Atomicity**: A transaction should be done or undone completely and unambiguously. In the event of a failure of any operation, effects of all operations that make up the transaction should be undone, and data should be rolled back to its previous state. ('A' should probably stand for "abortable".)
- **Consistency**: A transaction should preserve all the invariant properties (such as integrity constraints) defined on the data. On completion of a successful transaction, the data should be in a consistent state. In other words, a transaction should transform the system from one consistent state to another consistent state. For example, in the case of relational databases, a consistent transaction should preserve all the integrity constraints defined on the data. (Those who coined this acronym reportedly said the 'C' "was tossed in to make the acronym work"; consistency is a term with many meanings and was not considered important at the time.)
- **Isolation**: Each transaction should appear to execute independently of other transactions that may be executing concurrently in the same environment. The effect of executing a set of transactions serially should be the same as that of running them concurrently. This requires two things:
  - During the course of a transaction, intermediate (possibly inconsistent) state of the data should not be exposed to all other transactions.
  - Two concurrent transactions should not be able to operate on the same data. Database management systems usually implement this feature using locking.

This is an unsolved issue in databases (as of 2017). True or *strong* isolation is known as "serializable", but most implementations of that have major performance issues. So most DB vendors provide weaker isolation by default that doesn't provide the same guarantees, such as *read commited* or *snapshot* isolation levels. (And vendors don't generally mean the same thing when they use these terms!) Thus data loss and corruption do occur in the real world.

- **Durability**: The effects of a completed transaction should always be persistent. This too is trickier than it seems at first. If the disk reports the write was successful, was the data saved to non-volatile storage or just a RAM cache? If it was saved to disk, is that sufficient? If that replica dies (say due to a disk crash), the data is lost. Yet waiting for all replicas across an internet to report the data was saved takes too long. A compromise is to say it is durably saved if saved successfully to two replicas. Note, SSDs if unpowered will start to lose data in a few weeks, another aspect of durability to consider.

Strict, real, or true isolation is hard to implement well and expensive (takes a long time and other resources) to use. Most RDBMS provide a way to control the isolation level used, allowing one to trade strict isolation with better efficiency. While a system admin should rely on a database administrator (or the knowledgeable lead programmer of the code that will use the database), they will need to ask about this when setting up a database for them. So here's a brief overview:

The SQL standard defines four levels of transaction isolation. **The most strict is *Serializable***, which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in any order. All four were standardized in the 1990s and assumed that locks would be used to implement this. More recently, multi-version concurrency control (MVCC) supports lock-free concurrency, using *snapshot isolation*. Postgres supports a form of this, serializable snapshot isolation (SSI) which uses MVCC to improve performance, at the cost of more transactions failing to complete the first time.

Serializable isolation prevents the following potential issues from occurring: *dirty reads* (reading data modified by the transaction before the transaction is committed), *non-repeatable reads*, or *read skew* (a transaction often reads data more than once; if the data is different each time, that's a problem), *phantom reads* (similar to non-repeatable reads), and *write skew* (concurrent transactions each determine what they are

writing based on reading some data, which overlaps what another concurrent transaction is writing).

Repeatable Read isolation keeps read and write locks (acquired on selected data) until the end of the transaction. This isolation does not prevent phantom reads.

Read Committed isolation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the `SELECT` operation is performed. This may allow non-repeatable and phantom reads.

Read Uncommitted isolation is the lowest isolation level, allowing all the issues described above.

Unless serializable isolation (or in some cases, snapshot isolation) is used, it becomes the programmer's responsibility to use techniques to avoid any errors.These properties, known as ***ACID properties***, guarantee that a transaction is never incomplete, the data is never inconsistent, concurrent transactions are independent, and the effects of a transaction are persistent. (Most of the time anyway.) Never use a non-ACID DBMS for anything important!

The transactions are written to a *transaction log* file, sometimes called a *journal* or a *write-ahead log* (WAL). Once that write succeeds, the transaction is applied to the DB tables, with the changes also saved to a separate file. This way a failed transaction (from a crash or an abort) can be easily "rolled back" by undoing/discarding changes made from the aborted transaction.

DB transactions are similar to filesystem journaling. A DBMS records the commands for all changes in a given transaction to a log file (the journal). Then it makes the changes to the tables. Finally, the DBMS marks that log entry as complete. If the system crashes at any point, it can simply replay the journal entries to restore the DB, completing the transaction in progress. This works because such log entries are *idempotent*. That means running such commands more than once won't corrupt anything. (Most DBMSes also use a form of COW, meaning the DB tables look unchanged to other users until the transaction is complete.)

SQL Basics:

Most SQL statements are called *queries* or *updates*, and can be entered on one line or several lines. They end with a semicolon ("`;`"), although not all DBMSes will require this. The SQL keywords are not case sensitive; only data inside of quotes is case sensitive. (Some SQL DDL statements were shown above when defining a schema for normal forms.) SQL uses single quotes around literal text values (some systems also accept double quotes). Some examples of SQL:

```
INSERT INTO table (col1, col2, ...)
VALUES (val1, val2, ...);

SELECT [DISTINCT] col1, col2, ...
      or use wildcard:  * instead of a column list
FROM table [, table2, ...]
WHERE condition (e.g. WHERE amount < 100)
ORDER BY col;

UPDATE table
SET col2 = value2, col3 = value3, ...
WHERE condition; (e.g. WHERE col1 = value1)

DELETE FROM table
WHERE condition;     (e.g., WHERE col1=value1)
```

> A great way to practice and learn SQL is to use the SQuirreL SQL GUI client. This is a portable Java application (so you need to install Java first!), that is easy to use with any database. See SquirrelSQL.org to download or for more information.

Some of the more common *Data Manipulation Language* (DML) SQL statements (the ones used for CRUD) include: `SELECT` (to find and show data), `INSERT`, `DELETE`, and `UPDATE`. For example, here's a useful query done on the MySQL databased used for the class UnixWiki site:

```
SELECT user_name, user_real_name, user_email
 FROM unix_wiki.user;
```

Other SQL commands aren't as well standardized. They are used for defining schemas (Data Definition Language, or DDL) and for DBMS control operations (Data Control Language, or DCL). Some of the more common ones include: `CREATE`, `ALTER`, `DROP`, `GRANT` and `REVOKE`.

Other SQL commands aren't as well standardized. The ***Data Definition Language*** (DDL) is used for defining schemas (CREATE, ALTER, and DROP tables and views). The ***Data Control Language*** (DCL) supports DBMS control operations. Some of the more common DCL statements include GRANT and REVOKE.

> There is no standard SQL to **list the databases** (the SQL standard uses the term *schema*) available on some server.
>
> There is a standard SQL query to **list the schemas in a database** (the SQL standard uses the term *catalog*). (Note Some DBMSes don't support schemas, or don't follow the standard, e.g., DB2 and Oracle). For compliant RDBMSes use:

```
SELECT SCHEMA_NAME FROM
INFORMATION_SCHEMA.SCHEMATA
```

Not all RDBMSes support the standard SQL to **list the tables in a DB/schema**:

```
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
   WHERE TABLE_SCHEMA = 'name'
```

(Oracle uses "`SELECT * FROM TAB`"; DB2 uses "`SYSCAT`" instead of "`INFORMATION_SCHEMA`".)

Using `INFORMATION_SCHEMA` it is possible to describe (list the columns and their types and constraints) any table, but not all RDBMSes support this. For Oracle use "`DESCRIBE tablename`" and for DB2 use "`DESCRIBE TABLE tablename`".

A system administrator also needs to know a little about: **indexes**, ***views*** (virtual tables), ***sequences*** (generates the next number each time it is used), ***tablespaces*** (allows grouping of tables on the disk), ***triggers*** (do a task automatically when some condition occurs), ***functions***, and ***stored procedures***. While not all RDBMS systems support all these features, you do need to understand what they are. Here is an example using a sequence:

```
CREATE SEQUENCE seq;
INSERT INTO foo(id, name)
  VALUES(NEXTVAL('seq'), 'Hymie');
```

See one of the on-line SQL tutorials for more information; one of the best is sqlzoo.net, and the SQL tutorial at w3schools.com is pretty good too. I like the book The Manga Guide to Databases as an introduction, but I use on-line sources to learn SQL, especially when learning the non-standard SQL for some particular DBMS.

Prepared statements (and similarly for stored procedures) are SQL statements that are pre-compiled into an execution plan, and stored with the DB. These should always be preferred to regular (*dynamic*) SQL statements when possible, as they are more efficient and much more secure (mitigating against SQL injection threats).

Here's an example for MySQL prepared statements:

```
PREPARE stmt1 FROM 'SELECT id FROM Users
  WHERE name = ? AND password = ?';
SET @a = 'Piffl'; SET @b = 'secret';
EXECUTE stmt1 USING @a, @b;
```

And an example of creating and using a MySQL stored procedure:

```
DELIMITER $$
CREATE PROCEDURE getQtyOrders
 ( customerID INT, OUT qtyOrders INT )
BEGIN
   SELECT COUNT(*) INTO qtyOrders FROM Orders
   WHERE accnum=customerID;
END;
$$

DELIMITER ;
CALL getQtyOrders(1, @qty);
SELECT @qty;
```

Here's an example of a PostgreSQL prepared statement:

```
PREPARE usrRptPlan (int) AS
SELECT * FROM users u, logs l
  WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;

EXECUTE usrRptPlan(1, current_date);
```

And an example of creating and using a PostgreSQL stored procedure:

```
CREATE FUNCTION getQtyOrders(customerID int)
  RETURNS int AS $$
DECLARE
  qty int;
BEGIN
  SELECT COUNT(*) INTO qty FROM Orders
    WHERE accnum = customerID;
  RETURN qty;
END;
$$ LANGUAGE plpgsql;

SELECT getQtyOrders(12677) AS qty;
```

MySQL (and MariaDB)

The documentation for MySQL includes tutorial introductions and reference information, but a system administrator can get by with much less information, shown here. (Learning to install and configure a DBMS is covered elsewhere.)

To use MySQL or MariaDB may require some initial setup, depending on the version. For older systems, MySQL comes pre-configured with a root user without any password, and a test database any user can access. More recent versions come locked-down. After starting the server for the first time, you should run the command:

```
# mysql_secure_installation
```

This is a wizard that will ask some questions and un-lock the system for you. On the older versions, you need to set a password for `root@localhost`, and add a user or two. For each user, it is useful to create a separate database where they can experiment without affecting others. In addition, you might want to drop the `test*` databases. Accepting all the defaults from this screipt is fine.

You can avoid those command line arguments if you set the values in a file `~/.my.cnf`. This file can contain your plaintext password, but don't add that on a production server! In any case, set the permissions to 400.

Running MySQL from cmd line:

```
mysql -u user [-h host] -p
```

Changing passwords from the command line:

```
mysqladmin -u user password secret
```

The root user can also change passwords from inside the `mysql` DB:

mysql> **SELECT * FROM mysql.user WHERE user = "user"\G**
mysql> **SET PASSWORD FOR "*user*@localhost"=password('*new-password*');**
mysql> **SET PASSWORD FOR "*user*@%"=password('*new-password*');**

(Ending a MySQL query with "`\G`" instead of a semicolon results in a vertical output format, useful when there are many columns.)

Add users: (If using `mysqladmin`, run "`mysqladmin reload`" afterward.)

```
GRANT ALL PRIVILEGES ON *.* TO "user@localhost"
    IDENTIFIED BY 'secret' WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON *.* TO "user@%"
    IDENTIFIED BY 'secret' WITH GRANT OPTION;
```

Delete users:

mysql> `DELETE FROM mysql.user WHERE user = 'user';`

MySQL configuration and security model: define some users. Define a DB ("`CREATE DATABASE name;`"), add some tables, add some data, and do some queries. [Project idea: Design and implement a MySQL DB for an on-line greeting e-card site. (Show `HCCDumpSrc.htm`.)]

To recover a lost password, log in as root and change it as shown above. To recover a lost root password, stop the server. Create a text file `~root/reset-mysql-pw` with this **single command**:

```
UPDATE mysql.user
SET Password=PASSWORD('MyNewPass')
WHERE User='root'; FLUSH PRIVILEGES;
```

Then restart the server with:

```
mysqld_safe --init-file=~root/reset-mysql-pw &
```

When this has worked, delete the file (it contains a password), and restart the server normally.

It is possible, but dangerous, to start the server with "`--skip-grant-tables --skip-networking`", then login as user `root` and no password is needed. You can then reset anything, then restart as normal.

(Debian stores the clear-text password for its system MariaDB user in `/etc/mysql/debian.cnf`.)

You can dump MySQL/MariaDB databases as plain text files, containing the SQL statements needed to recreate the DBs. This has many uses, including systematic changes via some script, then re-creating the DB from the modified text. You can also use tools such as `grep` on this file, which can be useful when you don't know which table has some data you're looking for. The dump is a portable backup, which can be used with other DBMSes. To create an SQL text dump, run:

```
mysqldump --extended-insert=false --all-databases > dbdump.txt
```

(You can specify just one or a few DBs with the option "`--databases mydb`".)

PostgreSQL (a.k.a. Postgres)

PostgreSQL (which is pronounced post-gres-q-l) is often just called "Postgres", the original name before it switched to use SQL. Originally, it just used Unix system user accounts, so no extra effort was needed to add users. For modern versions (currently version 9) you must instead add a *role* for each user who is allowed to connect to the server.

The rules for authenticating users are controlled by a configuration file, `/var/lib/pgsql/data/pg_hba.conf`. The default for local (non-network) user access via the `psql` command line tool is to just trust them. So you should not need to use a password! (This can be changed to increase security.) Other choices include `pam`, `ldap`, `md5`, `ident sameuser` (often used for local access, this means allow a user to connect without a password using their system login name as the role name), and others.

In PostgreSQL, a *user* is really just a *role* with login ability. Creating users with `CREATE USER` is the same as `CREATE ROLE WITH LOGIN` (i.e., a role with login privilege).

The PostgreSQL security system is a bit simpler to understand than the MySQL system. With PostgreSQL, the *owner* of some *object* (e.g., a database or a table) can do anything to it, as can any PostgreSQL administrator user (a.k.a. *super-user*). All other users must be granted/denied access to objects using the SQL `GRANT` and `REVOKE` commands.

When adding user roles to PostgreSQL, use either the `CREATE USER` non-standard SQL command or the command-line tool **`createuser`**. With this tool you can also put in a password for the user. There is also a **`dropuser`** utility that matches the similar (non-standard) SQL command. (Use "`-e`" to see the generated SQL.) Roles are stored in `pg_catalog.pg_roles`. This table can be viewed, or modified to **change proprieties or passwords of users**:

```
\x
SELECT * FROM pg_catalog.pg_roles WHERE rolname='auser';
ALTER ROLE auser WITH PASSWORD 'secret';
\x
```

"`\x`" toggles vertical or horizontal output of rows. For displaying a single row with many columns, I prefer the vertical (or *expanded*) output format. Besides the password you can alter the other *role* properties too.

To work with PostgreSQL, you must configure `pg_hba.conf` (or live with the defaults), initialize the database system (done as part of the install; you only need do this step once), and then start it running:

```
# export PGDATA=/var/lib/pgsql/data
# cd /tmp
# su -c 'initdb' postgres
```

On Sys V init based systems, such as Fedora 15 or older, you can use:

```
# /etc/init.d/postgres initdb
```

For Fedora 16 and newer Red Hat systems, systemd has replaced Sys V init scripts and you can't do this anymore. Instead, run this new command:

```
# sudo postgresql-setup --initdb
```

The only admin (actually the only user) initially is "`postgres`". You need to start the server and create the user `root` within PostgreSQL. To start the server:

```
# /etc/init.d/postgres start  # SysV init system
```

To start the server using systemd, run this command instead:

```
# systemctl start postgresql.service
```

Next, make a new Postgres user "root", as an administrator:

```
# cd /tmp; su -c 'createuser -s root' postgres
```

The above command makes `root` an administrator too (the "`-s`" option). (Note that user `postgres` may not have access to your home directory. To avoid an error, you should `cd` into a public directory such as `/tmp` before running this.)

To allow the use of the PostgreSQL procedural language extensions to SQL, you must add this "language" to a database first. You can do this to the default template ("`template1`") which is copied when creating new databases:

```
# su -c 'createlang plpgsql template1' postgres
```

(This is done automatically on newer systems; if so, you will see a harmless error message.)

Finally we are ready to create a database for some user. First, you must add the username as a new *role* in postgres. Then you can create a database owned by that user, using SQL or the shell utility `createdb`. Note that `createdb` defaults to creating a DB named for the current user (and owned by that user). Since `root` has database creation privileges, note how simple the first example below is:

```
# createdb # Create a DB named root, owned by
root
# createuser auser
# createdb -O auser auser
```

(You can add a comment for your DBs with additional arguments.) To connect to the database as some user (other than your current UID), run this:

```
# psql -d auser -U auser
```

(Note if you log in as `auser`, then you can omit all the command line arguments, as that will attempt to connect to the `auser` database by default.)

You can also do these steps from within PostgreSQL. For example, to create a new user, run psql as a privileged user and enter this SQL:

```
username=# CREATE USER name WITH PASSWORD
password;
```

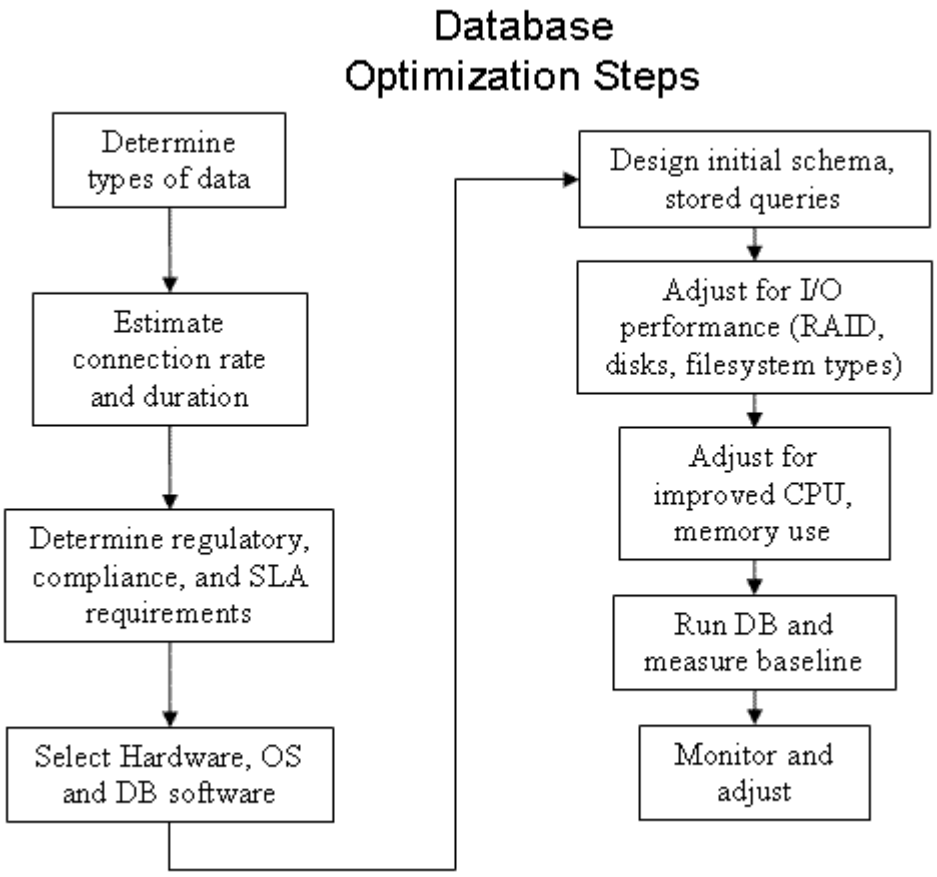
And to drop a user, enter this SQL:

```
username=# DROP ROLE user;
```

The last step is to enable your database server to start automatically at boot time. Again your init system dermines how this is done. With systemd, do this:

```
# systemctl enable postgresql.service
```

RDBMS Performance Tuning — *A Dark Art*



The first task is to make sure the database is running on appropriate hardware. Modern SCSI (such as SAS) may be best but SATA will work well too. Make sure it is modern or it may not support *write barriers* correctly. Use enterprise grade disks. Aside from spinning 2 or 3 times as fast as consumer grade ones, the firmware is better and more predictable (replacing a consumer grade disk, even with the same model, may not result in the same disk firmware.) Enterprise disks can also have non-volatile write caches (backed up with a battery or large capacitor).

Next, decide on central storage (SAN or NAS), or DAS. DAS may perform better (no network latency and no HBA, network switch, or OS write caches between your OS and the drives) but are not as flexible and may cost the same or more than central storage.

Turn off any OS LVM or software RAID. As for RAID, most database servers will work best with RAID 10. This give similar reliability to RAID-5 or RAID-6, but without parity calculations to slow down writes.

Finally, you will do best if you use the lower numbered cylinders (outer edge) of any spinning disks for the logs and database tables, and use the slower inner cylinders for OS and other system files. SSDs will work even better.

You need to monitor CPU usage (`mpstat`), RAM usage (`free` or `vmstat`), and disk I/O (`iostat`). (Linux tools; Unix may have different tools.) By monitoring these, you can see if a lack of hardware resources is the problem (and view trends over time, to predict when hardware upgrades might be useful).

Two key points for the SA are to **use a safe filesystem type** (e.g., ext4 or XFS, but not JFS or ReiserFS), and to **force disks to write data immediately** (known as ***direct write***) by turning off any hardware disk buffering/caching (via `hdparm -W 0`), unless using a non-volatile buffer. Turn off any kernel buffering too! (Otherwise the DBMS thinks the data has been written when it may not have been; so crash recovery may lose/corrupt data.)

> With ext4 filesystems, you can control the journaling feature: no journaling (*write-through* cache) with the mount option `data=writeback`, ordered writes (only filesystem metadata is journaled, but the by writing data then the metadata makes this fairly safe) with the mount option `data=ordered`, and journal (everything is journaled; safest but slowest) with the mount option `data=journal`. Best advice is to use journal mode and change to ordered mode only if performance is bad *and* changing this seems to make a difference.

Different types of filesystems work better with DBs than others. Using any FAT filesystem for a serious DB will cause poor performance. Using something like JFS or ext2 can improve performance at the cost of safety (not all writes are journaled). Using a journaling filesystem or RAID-1 (or similar) works best for DBs that don't do that internally anyway (but most do, with a transaction log file). Note that without filesystem journaling, the transaction log file can become corrupted in a crash. Using tablespaces, you can put that log on one filesystem and the data on another.

Depending on how critical the data is, you may not want to use direct write as it slows down access to other files on that disk volume. With a decent UPS the small safety gain may not justify the performance loss. On the other hand an important DB shouldn't share a storage volume with other files; it should have one or more storage volumes of its own. Also consider that with SAN/NAS/external RAID/JBOD storage systems, you may not have control over

this, and in any case there are many caches between the server's memory and the disk platter (the HBA, network switches, NAS head).

Creating a separate partition and filesystem just for the DB files works better than creating those files in (for example) the root file system with lots of non-DB files. Using separate filesystems for indexes, table data, and transaction logs can also greatly improve performance, especially if the different filesystems are on different disk spindles.

Either use a faster filesystem type and RAID-0 and let the DBMS handle those issues, or tune the DBMS to not bother with journaling (except for the logs) and/or mirroring.

Once the hardware is selected and your data is safe, then consider performance tuning. Poor performance will result when using a DB on a default filesystem type with default settings. A poorly tuned system can be much slower (10 times) than a properly tuned one! The different can be between minutes and days to execute some query. For example using Oracle on a FAT32 or ext4 filesystems with RAID-1 when Oracle uses different block and stripe sizes, journals writes to its files, and possibly mirrors the tablespaces, is not going to be fast!

First off, make sure you use a modern kernel version. Many have improved disk software a lot in recent years (2015). After choosing the file system type, set FS parameters.

Block (cluster) size can have a big impact on DBMS performance, particularly when running a database much bigger than system memory. One way to solve this problem is to reduce the value of the file system cluster size (`maxcontig` parameter) so that the disk I/O transfer size matches the DB block size. Another solution is to enable file system Direct I/O by mounting the file system with the right option (`--forcedirectio` for UFS), since file system Direct I/O will automatically disable the read-ahead. In addition, since MySQL has its own data and cache buffers, using Direct I/O can disable the file system buffer to save the CPU cycles from being spent on double buffering.

For most RDBMs the default OS value for disk *read-ahead* setting is too small, usually 256 (= 128 KiB on older drives). A good value should be 4096 to 16384. To set on Linux, use either **`hdparm`** or **`blockdev`** `--setra` (usually from `rc.local`). Next, mount the filesystem(s) with `noatime`. These two measures are probably the most important ones to improve performance.

The final OS tunable parameters to worry about are for OS caching and swapping. On Linux, set `vm.swappiness` to 0 (default: 60) to make the system avoid swapping as much as possible. (The trade-off is a smaller disk cache, but that's worthwhile for most DBs.) To also help prevent swapping, set

`vm.overcommit_memory` to 2 (never overcommit memory). Lastly you can control how many dirty memory pages the OS will allow to be outstanding before flushing them to disk; too many and the delay when the flush does occur will be noticeable, especially if you have lots of RAM. You can set `vm.dirty_ratio` to 2 and `vm.dirty_background_ratio` to 1 if you have more than 8 GiB of RAM. (All of these settings can be changed using `sysctl`, or directly using the `/proc` system. Make sure you set these at boot time, either by editing `sysctl.conf` or `rc.local`.)

One of the most important steps to tuning a DB is to **create a good set of indexes**. (This is likely more important than messing with tunable parameters.) Without proper indexes for you tables, many queries rely on sequential (linear) searching. Having too many indexes can also hurt performance, as the system must maintain all indexes whenever a table is updated. The primary key column(s) should be indexed (default for MySQL). Other columns may or may not benefit from indexing. By using the `EXPLAIN` command to show you details of the most common queries, you can determine which other indexes might help.

Bad performance can be the result of poor DB management: most DBMS use DB statistics (row counts, etc.) as input to the query optimizer. If these stats aren't automatically updated by the DBMS and you don't update them regularly, the optimizer can do a spectacularly poor job. For example, if the stats say a table with 100,000 rows has only 10 rows, then most optimizers would scan it sequentially because a sequential scan on a table that fits in one disk block will always be quicker than indexed access. You should turn on auto-stats (or whatever that feature is called on your DBMS) if that's an option, or manually update them from time to time otherwise.

Performance also depends on the amount of writes versus reads for your application. Filesystems are typically designed have reads as fast as possible, and the difference in read and write speeds can be very noticeable.

If using NAS or SAN, network congestion is another factor.

Many enterprise-class DBMS (e.g., Oracle) don't require a filesystem at all and can manage the raw disk space themselves. This feature is often referred to as "tablespaces", each of which can be thought of as a file holding the DB data and meta-data. (With PostgreSQL, tablespaces sit atop regular filesystems.)

Setting the DB cache sizes is important as well, or the files used for the DB will end up swapped to disk and the frequent dirty page writes will slow down the whole system.

Keep in mind monitoring and backups: If using a filesystem the DB gets backed up (and disk space gets monitored) by your normal filesystem tools and procedures. If using raw disk volumes then you must use your DBMS system to

monitor and backup the data using a separate procedure. Have an appropriate backup policy (SLA) and restore procedures. Often a monitor process must be kept running too, for security, compliance auditing, and baselining purposes.

The bottom line is for small or web site DBs that are mostly read-only, a filesystem based DB should be fine. For large OLTP systems, you need to have the DBA and the system administrator work together to tune the disk layout, the filesystem types used, and the DBMS itself. If using some enterprise DBMS such as Oracle that handles much of what the filesystem and RAID system can do itself, using a raw disk volume (and properly tuned DB) will result in the greatest performance for OLTP systems.

MySQL (and MariaDB) Specific Tuning

MySQL is a single-process, multithreaded application. The main thread is idle most of the time and "wakes up" every 300 milliseconds (msec) to check whether an action is required, such as flushing dirty blocks in the buffer pool. For each client request, an additional thread is created to process that client request and send back the result to each client once the result is ready.

MySQL includes several storage engines including MyISAM, ISAM, InnoDB, HEAP, MERGE, and Berkeley DB (BDB), but only **InnoDB** storage supports ACID transactions with commit, rollback, and crash recovery capabilities, and row-level locks with queries running as non-locking consistent reads by default.

InnoDB also has the feature known as ***referential integrity*** with ***foreign key constraints*** support, and it supports fast record lookups for queries using a primary key. Because of these and other powerful functions and features, InnoDB is often used in large, heavy-load production TP systems.

MyISAM (and ISAM) is for simpler applications (e.g., PHP blog, catalog data, and other non-OLTP applications). This has fewer features and limited scalability, but can provide superior performance (esp. connection speed) when you don't need those features. You can use different engines for different tables in the same database, to get the maximum performance and safety.

MySQL has peak performance when the number of connections equals roughly 4 times the # of CPU cores. By estimating the number of concurrent connections, you can plan how large a SMP or cluster to use (i.e., how many DB servers are needed).

MySQL doesn't access the disk directly; instead, it reads data into the internal buffer cache, reads/writes blocks, and flushes the changes back to the disk. If the server requests data available in the cache, the data can be processed right away. Otherwise, the operating system will request that the data be loaded from the disk.

Use EXPLAIN *query* to see what MySQL does. Use this insight to see where to change queries and/or add indexes. Tuning your queries and adding the required indexes is the best way to affect performance.

Once you've chosen the proper filesystem types, created the appropriate number of filesystems for your database, set their options and mount options correctly, and set the OS tunable parameters appropriately, **it is time to consider adjusting the database's internal tunable parameters**. Normally, that is left for a DBA to handle, not the system administrator (MySQL has over 430 tunable parameters!). But there are some performance measures you can consider for MySQL (and MariaDB):

- For applications where the number of user connections is not tunable (i.e., most of the time), the **`innodb_thread_concurrency`** parameter can be configured to set the maximum number of threads concurrently kept inside an InnoDB. (Other threads are kept waiting their turn.) If the value is too small under heavy load, threads will be kept waiting and thus performance will suffer. You need to increase this value when you see many queries waiting in the queue in `show innodb status` output. Setting this value at 1000 will disable the concurrency checking, so there will be as many threads concurrently running inside InnoDB as needed to handle the different tasks inside the server, but too many requests at once can also hurt performance.
- **`table_open_cache`** is the maximum number of simultaneous open files MySQL will request. Since each table is a separate file, you should set this to at least the number of tables used in your most complex query, times the number of DB connections you allow from clients. The default value (for MySQL 5.5) is 400 (used to be 64) and a real-word DB may have hundreds of tables, dozens at a time used in queries, with dozens to hundreds of open connections from clients. Note your OS must allow that many open files per process.
- The **`query_cache_size`** is the amount of memory to use to store the results of previous queries. This parameter can be set to 128MB or less, to provide better performance if usually many clients do read operations on the same queries. For some reason, this is set to zero on MySQL 5.6!
- The modern MySQL and MariaDB use the InnoDB back-end by default. The parameter **`innodb_buffer_pool_size`** determines how much memory the DBMS will request from the OS. On a DB server, you will get better performance from your system if you assign memory to this buffer pool, rather than a general OS page (disk) cache. The default value of 8 Mbytes is too small for most workloads. You will need to increase this number when you see that %b (percentage utilization of the disk) is above 60%, `svc_t` (response

time) is above 35 msec in the `iostat -xnt 5` trace output, and a high amount of reads appear in the FILE IO part of the `show InnoDB status` output. However, you should not set the cache size too large. If you do, you run the risk of expensive paging for the other processes running without enough RAM, and it will significantly degrade performance. For systems running a single dedicated MySQL process only, it should be fine to set this parameter up to a value between 70 and 80 percent of memory since the footprint of the MySQL process is only around 2 to 3 Mbytes.

- Another parameter to tune is **`key_buffer_size`**, which defaults to 1MB on older versions, and controls the size of the common cache used by all threads. With >256MB of RAM, set to 64M at least.

Scaling and Reliability

DBs are often a vital part of an enterprise, and must be highly available. This often means using some duplicate hardware in order to improve reliability (includes servers and disks). Duplicating the data on multiple servers is called ***replication***. Also *clusters* are sometimes used to provide ***transparent failover*** and ***load balancing***. In really large DBs ("big data"), tables are often split into ***shards***. Such a ***distributed*** DB requires the query to figure out which shard holds the data requested, or to query each shard in turn and merge the results. (Example: DNS data is distributed and replicated.)

> Most of these DB topics are beyond the scope of Sys Admin, certainly beyond the scope of this course. You are encouraged however to know these terms and concepts in a general sense.