***Notes on real-time renderers***

This accounts only for well-established methods, there are many other methods and combinations of methods I'm not covering. It's a sort of "recap".  
  
**Forward**  
Single pass over geometry generates "final" image, lights are bound to draw calls (via uniforms), accurate culling of light influence on geometry requires CSG splits. Multiple lights require either loops/branches in the shaders or shader permutations.

* Benefits
  + Fastest in its baseline case (single light per pixel, "simple" shaders or even baked lighting). [Doesn't have a "constant" up-front investment](http://timothylottes.blogspot.ca/2014/09/forward-vs-deferred-from-perspective-of.html), you pay as you go (more lights, more textures...).
  + Least memory necessary (least bandwidth, at least in theory). Makes MSAA possible.
  + Easy to integrate with shadowmaps (can render them one-at-a-time, or almost)
  + No extra pass over geometry
  + Any material, except ones that require screen-space passes like Jimenez's SS-SSS
* Issues
  + Culling lights on geometry requires geometrical splits (not a huge deal, actually). Can support "static" variations of shaders to customize for a given rendering case (number/type of lights, number of textures and so on) but "pays" such optimization with combinatorial explosion of shader cases and many more draw calls.
  + Culling dynamic lights can't be efficiently done. Scripted lights along fixed paths can be somewhat culled via geometry cutting, but fully dynamic lights can't efficiently cut geometry in runtime, just be assigned to objects, thus wasting computation.
  + Decals need to be multipass, lit twice. Alternatively, for static decals mesh can be cut and texture layering used (more shader variations), or for dynamic decals color can be splatted before main pass (but that costs an access to the offscreen buffer regardless or not a decal is there).
  + Complex shaders might not run optimally. As you have to do texturing and lighting (and shadowing) in the same pass, shaders can require a lot of registers and yield limited occupancy. Accessing many textures in sequence might create more trashing than accessing them in separate passes.
  + Lighting/texturing variations have to be dealt with dynamic branches which are often problematic for the shader compiler (must allocate registers for the worst case...), conditional moves (wasted work and registers) or shader permutations (combinatorial explosion)
  + Many "modern" rending effects require a depth/normal pre-pass anyways (i.e. SSAO, screen-space shadows, reflections and so on). Even though some of these can be faded out after a preset range and thus they can work with a partial pre-pass.
  + All shading is done on geometry, which means we pay all the eventual inefficiencies (e.g. partial quads, overdraw) on all shaders.

**Forward+ (light indexed)**  
Forward+ is basically identical to forward but doesn't do any geometry split on the scene as a pre-pass, it relies on tiles or 3d grids ("clustered") to cull lights in runtime.

* Benefits
  + Same memory as forward, more bandwidth. Enables MSAA.
  + Any material (same as forward)
  + Compared to forward, no mesh splitting necessary, much less shader permutations, less draw calls.
  + Compared to forward it handles dynamic lights with good culling.
* Issues
  + Light occlusion culling requires a full depth pre-pass for a total of two geometrical passes. Can be somehow sidestepped with a clustered light grid, if you don't have to end up splatting too many lights into it.
  + All shadowmaps need to be generated upfront (more memory) or splatted in screen-space in a pre-pass.
  + All lighting permutations need to be addressed as dynamic branches in the shader. Not good if we need to support many kinds of light/shadow types. In cases where simple lighting is needed, still has to pay the price of a monolithic ubershader that has to consider any lighting scenario.
  + Compared to forward, seems a steep price to pay to just get rid of geometry cutting. Note that even if it "solved" shader permutations, its solution is the same as doing forward with shaders that dynamic branch over light types/number of lights and setting these parameters per draw call.

**Deferred shading**  
Geometry pass renders a buffer of material attributes (and other proprieties needed for lighting but bound to geometry, e.g. lightmaps, vertex-baked lighting...). Lighting is computed in screenspace either by rendering volumes (stencil) or by using tiling. Multiple shading equations need either to be handled via branches in the lighting shaders, or via multiple passes per light.

* Benefits
  + Decouples texturing from lighting. Executes only texturing on geometry so it suffers less from partial quads, overdraw. Also, potentially can be faster on complex shaders (as discussed in the forward rendering issues).
  + Allows volumetric or multipass decals (and special effects) on the GBuffer (without computing the lighting twice).
  + Allows full-screen material passes like analytic geometric specular antialiasing (pre-filtering), which really works only done on the GBuffer, in forward it fails on all hard edges (split normals), and screen-space subsurface scattering.
  + Less draw calls, less shader permutations, one or few lighting shaders that can be hand-optimized well.
* Issues
  + Uses more memory and bandwidth. Might be slower due to more memory communication needed, especially on areas with simple lighting.
  + Doesn't handle transparencies easily. If a tiled or clustered deferred is used, the light information can be passed to a forward+ pass for transparencies.
  + Limits materials that need many different material parameters to be passed from geometry to lighting (GBuffer), even if shader variation for material in a modern PBR renderer tends not to be a problem.
  + Can't do lighting computations per object/vertex (i.e. GI), needs to pass everything per pixel in the GBuffer. An alternative is to store baked data in a voxel structure.
  + Accessing lighting related textures (gobos, cubemaps) might be less cache-coherent.
  + In general it has lots of enticing benefits over forward, and it -might- be faster in complex lighting/material/decal scenarios, but the baseline simple lighting/shading case is much more expensive.

**Notes on tiled/clustered versus "stenciled" techniques**

On older hardware early-stencil was limited to a single bit, so it couldn't be used both to mark the light volume and distinguish surface types. Tiled could be needed as it allowed more material variety by categorizing tiles and issuing multiple tile draws if needed.

On newer hardware tiled benefits lie in the ability of reducing bandwidth by processing all lights in a tile in a single shader. It also has some benefits for very small lights as these might stall early in the pipeline of the rasterizer, if drawn as volumes (draws that generate too little PS work).   
  
In fact most tile renderers demos like to show thousands of lights in view... But the reality is that it's still tricky to afford many shadowed lights per pixel in any case (even on nextgen where we have enough memory to cache shadowmaps), and unshadowed, cheap lights are worse than no lighting at all.  
  
Often, these cheap unshadowed lights are used as "fill", a cheap replacement for GI. This is not an unreasonable use case, but there are better ways, and standard lights, even when diffuse only, are actually not a great representation of indirect radiance.   
Voxel, vertex and lightmap bakes are often superior, or one could thing of special fill volumes that can take more space, embedding some radiance representation and falloff in them.  
  
In fact one of the typical "deferred" looks that many games still have today is characterized by "many" cheap point lights without shadowing (nor GI, nor gobos...), creating ugly circular splotches in the scene.   
Also tiled/clustered makes dynamic shadows somewhat harder, as you can't render one shadowmap at a time...

Tiled and clustered have their reasons, but demo scenes with thousands of cheap point lights are not one. Mostly they are interesting if you can compute other interesting data per tile/voxel.  
You can still get a BW saving in "realistic" scenes with low overlap of lighting volumes, but it's a tradeoff between that and accurate per-quad light culling you get from a modern early-z volume renderer.

**Deferred lighting**  
I'd say this is dwindling technique nowadays compared to deferred shading.

* Benefits
  + It requires less memory per pass, but the total memory traffic summing all passes is roughly the same. The former used to be -very- important due to the limited EDRAM memory on xbox 360 (and its inability to render outside EDRAM).
  + In theory allows more material "hacks", but it's still very limited. In fact I'd say the material expressiveness is identical to deferred shading, but you can add "extra" lighting in the second geometry pass. On deferred shading that has to be passed along using extra GBuffer space.
  + Allows shadows to be generated and used per light, instead of all upfront like deferred shading/forward+
* Issues
  + An extra geometric pass (could be avoided by using the same GBuffer as deferred shading, then doing lighting and compositing with the textures in separate fullscreen passes - but then it's almost more a variant of DS than DL imho)

*Some Links:*

* [*http://developer.amd.com/tools-and-sdks/graphics-development/amd-radeon-sdk/#downloadsamples*](http://developer.amd.com/tools-and-sdks/graphics-development/amd-radeon-sdk/#downloadsamples)
* [*https://software.intel.com/en-us/vcsource/samples*](https://software.intel.com/en-us/vcsource/samples)