Physical Database Design

The logical design of a database, usually expressed as a set of relations (SQL tables), still leaves many choices out as to how it is actually implemented in a computer. The process of making those decisions is called **physical database design**.

The most important issue to realize is that the physical design affect mostly **performance**; except for a very few issues, like the choice of data types for attributes, which may also affect the range of values that may be represented.

In most cases, performance in database applications is dominated by disk accesses; if we can minimize the number of disk accesses, even at the cost of using more CPU or RAM, we will almost always get better performance.

Although we conceptually consider this activities design, and we put them before implementation in our development life cycle, it is important to realize that many of these decisions can be taken during or after implementation and so could also be considered database maintenance. Deferring the activities until after the system has been implemented allows for better decisions, since the performance effects can be directly measured rather than estimated.

1. Decisions taken during Physical design

There are several decisions that need to be taken during physical design:

1. **Data types for each attribute**. We need to choose the data type for each attribute; in making this decision, we need to make sure the data type chosen can represent all desired values, and that it provides the best performance (uses less space and requires less disk accesses).
2. **Physical Record Descriptions.** When storing tables on disk, we use the term physical record rather than row. In the simplest model, each record is of the same size; although, depending on the data types, records of the same type may have different sizes. In the vast majority of cases, the physical record directly corresponds with the row; that is, the physical record contains the same fields as the row.
3. **Physical File Organization.** File organization refers to how are records organized in a file; meaning do we sort the records, write them as they come or use some other mechanism ?
4. **Indexes**. Indexes are data structures that make some searches faster, although they make updates to the database somewhat slower, since the indexes would need to be updated too. Indexes usually can only speed up searches on certain fields, and there are many different kinds of indexes available; so the decisions are which kinds of indexes to use, and on which fields.

2. Inputs to the Physical Design stage

The main inputs that the physical design stage requires are:

1. Normalized relational schema and Attribute definitions – which would allow us to define the records and the right data types for each of the fields.
2. Volume estimates - Both for the sizes of each relation and the number of queries of each different type. Notice we do not need perfect estimates, ballpark figures are usually enough.
3. Other needs – Including the choices of DBMS and hardware (or the possible choices), response time expectations, security, backups, recovery, integrity etc; although clients will always want the highest levels of these issues, there's an obvious tradeoff with cost, and so the needs will vary depending on the importance of the system.

3. Basic Concepts

When designing the physical structure of a database, we define a **field** as the smallest unit of data, and a **record** (or physical record) as a group of fields stored together and retrieved as a unit.

Storage Hierarchy

Since databases are repositories of data, when designing database systems we need to consider storage. When considering storage technologies, we usually have an inverse relation between price and speed; the faster the storage, the more expensive it is. This leads us to have different kinds of storage, organized in a hierarchy. A simple classification of storage is:

- **Primary storage** is the one directly accessible to the CPU; normally, RAM. Most of the primary storage is **volatile**, that is, it is erased when the computer is shut down.
- **Secondary storage** is not directly accessible to the CPU, but is directly accessible by the computer; the most common form of secondary storage is the hard drive.
- **Removable storage** (also called **tertiary storage**)is not directly attached to the computer, like tapes etc

1 Hard disk drives

Most databases cannot be completely in main memory, and they need to be stored in hard drives. Probably their most important characteristic is the fact that they can't read byte by byte, but they always read a fixed-size chunk, called a sector.

A hard drive has a number of platters, each platter divided into concentric **tracks**; the same track for all platters is oftentimes called **a cylinder**. Each track is further divided into **sectors**. A hard drive has one **read head** per platter, and the head can read one sector at a time; this means that to read a particular sector the drive needs to move the head up or down the platter, and also rotate the disk until the sector is below the head, which makes some sectors slower to access than others, depending on where the head is at any moment.

The most important issue with hard drives is that they are designed NOT to give access to individual bytes, but to only allow direct addressability of bigger chunks, usually called **pages**, which are on the order of one to a few kilobytes. Both the hardware on the hard drives, and the formatting given to them by an OS can affect the size of these pages, but the limitation always exists; even when your system calls allow you to read one byte, the full page needs to be transferred from the drive to the memory.

2 Fields

A field is the smallest unit of data stored in a database. When designing a field, the main issue is which data type we will use for it. Additionally, we may consider if we want to use some coding, compression or encryption on the field.

Each DBMS usually supports most of the standard SQL data types, plus a host of its own specific data types; many of the techniques used for the data types are similar from one DBMS to another, so we will discuss the techniques in the abstract. Keep in mind that each DBMS may apply the techniques to different data types.

For character, or string data types, the most important semantic difference is whether they are fixed-length (CHAR in SQL) or variable length (VARCHAR). For CHAR fields, if inserting a shorter value, the field will be padded with spaces. Each DBMS will decide how to actually implement these on disk, but for variable length records, the length of the value needs to be stored for **each** record, which may add a byte or two. another option (used by some DBMSs for really long text fields) is to store the text field separately, and just store a pointer to that text in the actual record.

For numeric fields, the implementation can use binary encoding, or some kind of decimal encoding, with slight differences in performance and semantics; a bigger difference would be if your DBMS supports floating point numbers, which save space at the expense of accuracy.

3 Physical Records

A **record** is a group of fields that are stored together and retrieved as a unit. It roughly corresponds to the concept of a row in the relational model. A file on disk will usually store records of the same type. Record types can be fixed-size, if all records are the same size (which implies all fielda are fixed-length) or variable sized.

Given that hard drives read a block at a time, we normally do not want a record to span two blocks, although several records can be stored on the same block; the number of records of a given type that fit in a block is called the **blocking factor** of that type (for variable sized records, we talk about the **average blocking factor**.

4 Denormalization

Sometimes, for performance reasons, we may need to undo the normalization of our tables, and store the data in unnormalized ways; the advantages of normalization usually far outweigh any performance gains, so we recommend **always** normalizing your tables.

4. Indexing and Hashing

Searching all the records in a file can take a long time; to avoid this, DBMSs provide for the creation of **indexing structures**, which help with searching; the most common indexing structures are B-trees, which are multi-way search trees.

You probably remember search trees, from your programming or data structures classes; when balanced, they allow for fast and efficient searching (but updates to the tree may need to rebalance it). For databases, we probably want to make nodes in the tree as big as possible, as long as they fit within a page on the disk; that is why we use multi-way trees in databases.

Figure 1 shows a balanced, binary search tree. It is called binary because each node has at most two children. For this tree we show only the keys (not the pointer to the record which will be associated with the key). In a binary search tree, nodes with lower keys will be on the left subtree, and nodes with bigger keys will be in the right-side subtree.

So, if we were looking for, say, the number 16 on the tree on figure 1, we would go to the root, which has a key of 17, since 16 is smaller, we'd then go to the left subtree, to the node with key 10; since 16 is bigger than 10, we'd then go to the right-side subtree, with key 14, and since 16 is bigger than 14, we'd again go to the right, and arrive at 16.

Now, if we assume information is only stored at the leaves (the last level), and the other keys are there only to facilitate the search, then we'd have 8 nodes with information on that tree; in order to find the information on that tree, we'd need only 4 comparisons, which is nice; however, the real power comes with larger numbers; if we add one more level, we get 16 nodes, and we'd only need 5 comparisons; one more level gives us 32 nodes but only 6 comparisons; as we keep adding levels, we double the amount of information, but only increase the number of comparisons by 1; so, 10 levels gives us 1024 nodes and 11 comparisons; 20 levels gives us about 1 million nodes and we only need 21 comparisons;

30 levels is about a billion nodes, and 40 levels is about one trillion nodes, but we'd need only 41 comparisons; binary search trees have *logarithmic* complexity on the number of comparisons; the number of comparisons is roughly the logarithm (base 2) of the number of nodes.

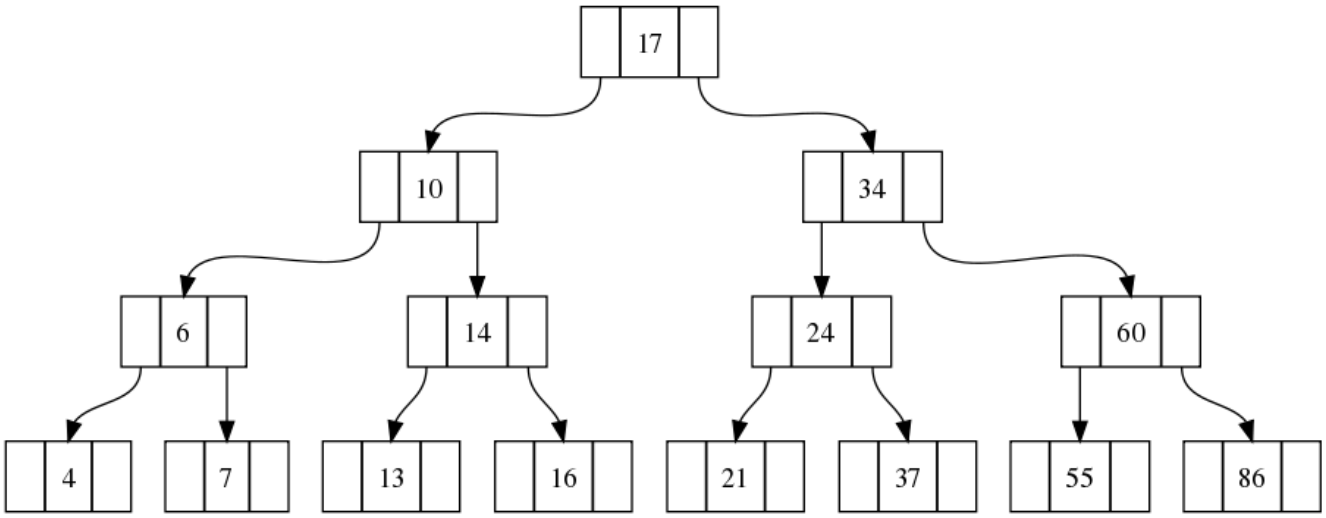

Figure 1: A balanced, binary search tree

Notice that for this to work, the tree needs to be balanced; unbalanced trees, like the one on Figure 2, do not help at all and may even slow down the searches. There are many techniques for balancing and keeping search trees in balance.

Now, if binary trees help, how about having a node have more children ? turns out this helps, but not much; it changes the base of our logarithms, so if we use, say 8 children, then we'd have, for a full tree of say, 3 levels, 8^3 nodes, instead of 2^3, since 8 is 2^3, then $8^3 = 2^{3x3} = 2^9$, so we'd need 9 levels for a binary tree, so we get a factor of 3 reduction; not bad, but not worth the extra complexity, since this is a constant factor.

When going to files, our main unit of work is the page, so we normally have multi-way trees, with as many children as needed so a node fits in a page with as little extra space as possible.

Another common indexing structure is a hash table; records (or pointers to them) are stored in different areas, called **buckets**, depending on the value of a **hash function** which is applied to the search key. Hash tables are normally efficient, but they may become very slow due to **collisions**, that is, too many records falling in the same bucket; the most efficient ways to solve this problem involve creating a new hash table for buckets that are too full, which makes the tables similar to tree-based indexes.

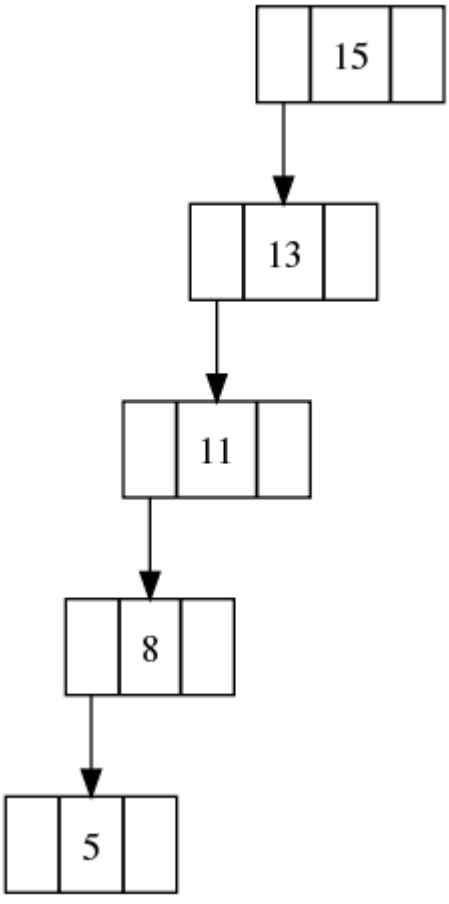

Figure 2: An unbalanced binary tree

In SQL, you can create indexes with the CREATE INDEX command, and most DBMSs provide for at least B-trees and most times several other kinds of indexes.

The important tradeoff that indexes create is that they make **searches much faster** (as long as we're searching on a field that has been indexed), but **updates somewhat slower**, since any updates to the database need to also update the index; also, indexes occupy extra space in the database.

5. Query Optimization

SQL is a **declarative** language, in that we specify **which kinds** of records we want, but we do not tell the DBMS how to retrieve them; the DBMS could retrieve in many different ways. Most DBMSs try to **optimize** a query, that is, to find the fastest way to obtain the results. In the most recent optimization paradigm, called **cost-based optimization**, the DBMS generates different **query plans**, that is, different ways in which it could go and calculate the desired results, and then tries to estimate the cost (usually the time) of that query; the DBMS will then choose the least expensive plan and execute it.

The important idea is that the SQL does NOT directly specify the algorithm, and so it is not always easy to see how to write faster queries; most DBMSs have a command called EXPLAIN that will give information about the query plan that it estimates is the fastest, for a given query.

6. RAID

RAID stands for Redundant Array of Inexpensive Disks, and is a set of techniques for using inexpensive disks to achieve much higher reliability or speed than the individual disks themselves. Many database servers (the actual machines) use RAID to obtain better performance.

Although a full discussion of RAID is outside the scope of this book, the main ideas are simple and will be explained here.

Imagine that you go to a store and buy a relatively cheap hard disk; now imagine that you want one that's twice as fast; chances are that it will be much more than twice as expensive; if you want a disk that's much more reliable, chances are that it will also be much more expensive; RAID allows you to combine several disks to achieve higher speed and/or reliability.

The way to achieve higher speed (or at least higher throughput) is to make the disks work in parallel, using a technique called **striping**. It consists of creating a virtual disk, where the pages come alternatively from one of several disks; so, if we have two disks, we can have it so the odd-numbered pages come from the first disk, and the even-numbered ones from the second disk; if we need to read, say, pages 2,3,4 and 5 from the virtual device, this means we read pages 2 and 4 from the second drive and 3 and 5 from the first one; since each drive is working separately the read operation will only take as much time as it takes each disk to read *two* pages, rather than four, doubling our throughput. Notice this only doubles throughput, not seek time, so if we need to read only one page, it still takes the same amount of time; it also adds a little overhead (since the controller of the virtual drive has to decide from which one to read etc), but the overhead is negligible, especially if the controller is implemented in separate hardware (a **RAID card**).

We achieve higher reliability by using **mirroring**, we use two disks, and store the *same* information on each drive; now if one disk fails, we have the other copy, and for the virtual disk to fail we need *both* disks to fail at the same time (or close enough that we haven't fixed the first failure). When writing, we need to write to both disks, but, since each disk work independently, this just adds a small overhead, and similarly for reading. Mirroring comes at a high cost, though, since we now need two drives to store the same information; other RAID levels use **parity** information, instead of keeping a full copy, which still allows us to recover from one disk failing, but allows for better utilization of the disks; the most commonly used RAID levels are 0 (just striping),1 (just mirroring), 0+1 (both mirroring and

striping, also sometimes known as RAID 10) and RAID 5, which uses block level parity and distributes it across all the disks. RAID 5 is almost as fast as RAID 0 for reads, but has much worse write performance.

A good (and more detailed) reference for RAID is http://en.wikipedia.org/wiki/Standard_RAID_levels