

By now you should know how to represent information in relational databases using SQL. SQL is great, but it is still a relatively high-level language; we talk about tables, but do not specify *how* these tables are going to be stored in the disk. Tying it to our introductory concepts, we're working at the *logical* level, but we still need to get to the *physical* level.

Notice that for program-data independence to work, the logical level should *not* need to change, regardless of how we physically organize the information; so the main difference that the physical level makes is *performance*. This is why we can live with *not* telling SQL about the physical design, and letting our DBMS choose by itself; the choice may not be the best one, but it will *work*; however, if we make the right choices, we can make it work much faster.

Keep in mind that SQL does not standardize physical design, so every DBMS has a different set of facilities supported, and the specific syntax we use to specify the physical design will vary considerably from DBMS to DBMS; because of this variations, we will cover the general concepts rather than covering the syntax on any specific DBMS.

Another issue is that, since the logical level will not change, we can defer these issues until after we have implemented the database; that is, we can do it as *maintenance*, rather than design; the advantage of deferring this process, is that we will have more information; we can actually test the speed when we choose one way of organizing the files over another, and we can test it on the actual hardware the system will be running on; a disadvantage may be that we may end up with a system that is too slow and may lose the trust of the users.

The slide has a green hexagonal pattern header. The title "Introduction" is in large, bold, black font. Below it, a list of bullet points is in green font. The SPSU logo is in the top right corner. A small number "2" is in the bottom right corner.

- DBMS needs extra, detailed info about how to store tables
- Main (only?) issue is performance
- **Minimize disk access**
- Logical design should NOT change !
- Many decisions can be deferred until implementation/maintenance
- Highly DBMS dependent

The main decisions you take when doing the physical design are the data types and generally the representation of your attributes; how to group fields in physical records, how to organize the records within a file, and whether to use indexes, and on what fields. We will cover each of these issues in more detail later.

Main Decisions

SPSU

- Attribute Data types
- Fields and Physical Records
- File Organization
- Indexes

3

Now, the inputs to the physical design process are the normalized relations, with the definition of each attribute, and volume estimates, both of the number of rows on each table, and the number of queries of each type that will be performed on the system. These estimates do not need to be precise, although the more precise the better (and deferring until the system is in production will guarantee much better estimates).

Besides these main inputs, there are other issues that we take into considerations.

One issue would be the response time expectations; although users would probably want the system to be as fast as possible, there's a tradeoff between speed and cost; you as the technical guide, would provide the tradeoffs and the users would make the decisions.

Another issue is the integrity expectations; again, most users would expect the system to work as specified, and to keep the integrity of data; however, high integrity might require faster, more expensive or redundant systems, increasing the cost; again, we provide information to the user, who makes the decisions; the integrity expectations of a banking systems may be much higher than, say, a blogging system, and so we will build more redundancy for the banking system.

We also have to worry about backup and recovery; An easy way to protect ourselves from system failure is to make a *backup*, a snapshot of the system, in a good state, at a moment in time; if the system fails later, we can *restore* that backup and get the system to that earlier good state; however, we lose all the work that was done between the backup and the restore; we also have the system down for the period of time it takes to restore the system, and may need some downtime to make the backup. The simplest and cheapest solutions will have more down time.

Another big issue is security; if we require more secure systems, we may need to get into specialized solutions.

Sometimes we get to choose which DBMS to use, but other times the choice is made by the client

Inputs

SPSU

- Normalized Relations and Attribute Definitions
- Volume estimates
- Other Issues
 - Response time expectations
 - Backup/Recovery/Security needs
 - Integrity expectations
 - DBMS etc

4

because of non-technical factors; the company may have more expertise with a given DBMS or may have done a strategic decision to use it on their future systems.

When designing computer systems, we usually have different kinds of memory; *primary* memory is directly accessible to the CPU; it basically consists of RAM; *secondary* memory is not directly accessible to the CPU, and it usually consists of hard drives; *tertiary* storage is removable, but there's a robotic mechanism that will insert the removable media (the typical example would be a tape library); *off-line* storage is removable and requires human intervention, like CDs or tape drives.

Usually there's a speed difference between kinds of storage; RAM is about 100 times faster than hard drives; there's also a price difference; RAM is about 10 times more expensive than drives; this usually means we have more of the cheaper memory, and use several levels of buffering or *cacheing*.

For example, we may have a machine with, say 4GB of RAM, and a 400GB hard drive; our database may be too big to fit in RAM; so modifying it may entail bringing the page into RAM, changing it, and bringing it back to the disk; we may want to keep some pages in RAM all the time, to improve performance, since reading from disk is slow, so an important issue for DBMS performance is how much memory to allocate to these *buffers*, which have a function similar to caches in a CPU.


Notice there is also qualitative differences; primary memory, RAM, is usually volatile; that is, the data will be erased when the system loses power, so data needs to be written to secondary memory; tertiary memory is removable, which makes it a better choice for backups; if the tape is in my safe, then a power surge may bring the server down, but I still have the info; a fire in my server room cannot destroy my tape if it is in a different room or building.

See http://en.wikipedia.org/wiki/Computer_data_storage for more about kinds of storage, and your favorite computer store for current price and performance characteristics.

Most of the databases will end up residing in a hard disk. The main issue to understand about hard disks is that the information is transferred to RAM in *pages*; if you want to read one byte from a disk, the hardware will still bring a full page into RAM; and when writing to the disk, the full page will be written; this means that we want to make sure our records do not overlap more than one page, so that we read at most one page for one record; we also want to fit as many records as possible into a disk page, so we need to read fewer pages in order to read all the information in a table.

We usually consider disks to be random


Memory Hierarchy



- Many different kinds of memory
 - Primary - RAM etc
 - Secondary - Hard Drives, now Flash/SSD
 - Tertiary - Removable, tapes
- Tradeoff price vs speed (for same size)
 - So we have more of slower kinds of memory
- Also, qualitative difference
 - Primary is volatile
 - Secondary is permanent, non-removable
 - Tertiary is removable

5

Hard Disks



- Most databases reside in hard disks
- Hard Disk issues
 - Info is transferred in **pages**
 - Slight difference in access time to reach each page, but considered 'the same'
 - Speed
 - Seek time
 - Transfer rate
 - About 100 times slower than RAM
 - The less pages I read/write, the faster my database

6

access devices, which is not quite true; we can access some pages faster than others; however, since the difference is small, we will usually discount it and think of it as if all pages can be accessed in the same amount of time.

When we talk about the speed of a disk, we are really trying to combine several different measures. The *disk access time* of a disk is how long it will take it to start reading a page; many times, we want to read several 'contiguous' pages in a row; the *transfer rate* is how many contiguous pages the hard drive can read in a given amount of time. Although the actual speeds of disks and RAM vary over time, transfer rates are about 100 times slower for hard disks, and the access time is at least a thousand times slower.

Some devices are *sequential*; to get to a given byte, the hardware needs to read *all* the bytes that are before it; the typical example would be a tape drive. This is ok when we want to access *all* the information on a device, but bad when we're trying to read only some information and it is not at the beginning. This makes tapes used mainly as backup media.

As mentioned above, on *paged* devices, like hard disks, we want to make sure the records do not go across page boundaries, which means we will have empty space on some pages. On sequential devices we do not want to do this, and may want to just write the records one after another; if the records are all of the same size, we do not even need a separator, and can just write the records one after another; if the records are variable sized, then we use a sequence of bytes that cannot appear in a record as our separator.

Notice that we sometimes use sequential files even on disks; for example, when writing plain text files, each line would be a record, and we usually write them one after another, using a newline character (or something like it) as separator.

One of the important issues in physical DB design is the design of each *field*; a field is the value stored for one attribute of one row on a table.

There are two main issues when considering how to represent each field on a table; the first is the data type that we will use to represent the field; the other one is if we want to do anything special with the value, like creating a code for it, using compression, or encrypting the field.

Sequential Files and Devices

SPSU

- Some devices (like tapes) are *sequential*
 - They have to be accessed in sequence
 - To get to byte 10000 I first need to read the previous 9999.
 - Slow access, but tapes are somewhat cheaper than disks, and they are removable, so better for storage
- Sequential files
 - Records are stored in the file one after another, instead of in pages
 - If records are variable size, need to use a separator
 - If same size records, can store directly one after another

7

Fields

SPSU

- Field:
 - The value in an attribute of a table
 - Basically a *cell* in the table
- Issues
 - choosing data type
 - do we use coding, compression or encryption ?

8

SQL stipulates a set of datatypes that should be provided by every DBMS, but each DBMS may choose to implement it differently, so the performance implications of the choice are heavily dependent on our DBMS; here we will describe the basic concepts, and you should consult your DBMS manual to verify how they implement each type. In addition, many DBMSs will support other data types not included in the SQL standard, and you may choose to take advantage of some of those.

Notice that the choice of data type may affect the semantics of the data, not just its performance, although we should be careful to achieve the desired semantics.

When choosing a data type for a field, we should try to choose one that takes the least amount of space, or the least number of bytes, while still being able to accommodate all possible values.

Now let's look at some of the issues when deciding which data types to use.

First, when representing string data, we have two standard options; we can use a CHAR or a VARCHAR field; here, there's a small semantic difference; if we insert a short string into a longer CHAR field, spaces will be added at the end, to make sure it has the appropriate number of characters; if the field is a VARCHAR, then no spaces will be added.

Now, the DBMS has three obvious ways to implement string types, which could be appropriate depending on whether the field varies and its length. First, it could always use the maximum amount of characters in the field, which would work well for CHAR fields, but could be wasteful for VARCHAR fields; also, it could store the length of the field, and then the exact number of characters contained in the field, which could work better for variable length strings; finally, it could store the actual characters on a different page, and store a *pointer* to those characters in the record, which could make retrieval of that record much faster, if that field is not needed.

When storing numeric fields, SQL supports fixed-precision numbers with the NUMERIC datatypes; most DBMSs also support standard integers in a variety of sizes, and standard floating point numbers.

The support for DATE fields in DBMSs varies widely; most DBMSs support at least two data types, one with a granularity of days, and one with a granularity of milliseconds, or even smaller, which is called TIMESTAMP on many systems. Of course, we need more bytes to store a TIMESTAMP.

Many times we want to store binary data, like an image or a sound in a database; most DBMSs support a type for Binary Large Objects, or BLOBs. The typical implementation would store the actual bytes in a separate page, and store a pointer to it within the record.

Fields: Choosing data type

SPSU

- DBMS Specific
- Affects semantics and performance
- Use as small (in bytes) a field as possible, but needs to accommodate all possible values
- Issues
 - CHAR vs VARCHAR vs Text/Long
 - NUMERIC vs int, float etc
 - DATE, TIMESTAMP
 - Binary Large Objects (BLOBs)

9

Many times we want to store string values, but we have a limited set of allowed values; in these cases, we can perform *coding* on the field; rather than storing the actual values, we store a much smaller code, and we create another table that maps that code to the values.

For example, assume we want to store information about people, including their gender, as in the table on the left; we could store the actual words 'Male' or 'Female' for gender, but this takes up to 6 bytes; instead, we can store only one letter, M or F, and create a new table that maps that code to the actual values, as in the two tables on the right.

In a way, this is equivalent to transforming the attribute into an entity in our ER model, with the code being its identifier, and having a relationship joining the original entity and this new one.

Coding can save space, but it needs an additional lookup, basically a JOIN, for when we want the actual value; however, since the table for the codes could be kept in memory, this could actually be faster, since we now read less bytes for each record; if queries now become more complex, we can always create a VIEW that has the actual values rather than the codes.

Notice that coding can also improve integrity, since referential integrity will ensure that values come from a specific set; it can also increase flexibility; if we need to allow a different set of values, we just change the table for the coding.

A *record* is a group of fields stored together in a file and retrieved as a unit; basically they correspond to the concept of a row, but at the physical level.

As we had mentioned before, disks cannot read individual bytes but, instead, they read larger blocks called *pages*. We do not want a record to span more than one page, since that would make it slow to retrieve; however, we will normally have more than one record on a page; usually, pages contain records of the same kind only.

The number of records of a given kind that fit in a page is called the page's *blocking factor*; we normally want our records to occupy as little space as possible, so more of them fit on a page. Notice that there's usually empty space at the end of a page, since we may not be able to fit an exact number of records on a page. Also, if the records vary in size, the blocking factor will vary, and so we would speak of an *average* blocking factor.

Again, notice that we can design *sequential* files, which just store the records one after the other, without worrying about pages; but these would be inefficient in hard disks as many records would span more than one page.

Fields: Coding
SPSU

- Uses a reference instead of actual value, and puts values in another table
- In a way, transforming an attribute into an entity in the ER diagram
- Saves space but needs additional lookup.
- Can improve integrity

Id	Name	Gender
1	Orlando	Male
2	Lina	Female
3	Jason	Male

Id	Name	Gender
1	Orlando	M
2	Lina	F
3	Jason	M

Code	Name
M	Male
F	Female

10

Physical Records
SPSU

- Record:** Group of fields stored adjacently and retrieved as unit (~row)
- Page:** Amount of data read or written in one IO operation (device dependent)
- Blocking factor:** # records per page
 - may vary !
 - records shouldn't span 2 pages
- Smaller records mean less pages**
- Sometimes, we use *sequential* files, but they are slower on disks**

11

Sometimes, we may think that normalizing our tables is making our database slow or complicated, and that we can duplicate information on certain tables, like keeping a field that we could obtain through a JOIN, or even keeping calculated fields; we call this process *denormalization*; usually, denormalizing our tables will lead to out-of-sync and wrong data, wasted space and hard to find errors in an application, so we strongly recommend always normalizing your tables.

Denormalization

SPSU

- **DON'T**
- Transforming normalized rels into unnormalized ones
- Possible benefits: less joins, faster
- Costs: Confusing !, wasted space, integrity threats

12

Some times, to improve performance, we can *partition* a table, and store it in more than one file.

If we store full rows in each file, but we store different rows in different files, we are doing *horizontal* partitioning; for example, if we have a student table, we could store the students for each major in a different file; so all the CS students in one file, all the IT students in another and so on; if the files are on different drives, we can increase parallelism and so performance.

If we store pieces of the rows in different files, we are doing *vertical* partitioning; we need to have the primary key in both files, or some other way to join the corresponding rows back together, so we're using more space; in vertical partitioning the speed comes from the fact that, if we're retrieving only the fields from one partition, then we only need to read those records, and not read anything from the other partition; however, if we need to retrieve fields from both partitions, then we'll need to do additional work to join the rows on both partitions.

We can also combine the partitioning mechanisms, and partition the same table both horizontally and vertically.

Partitioning

SPSU

- Horizontal: Distribute rows into several files
- Vertical: Divide rows, put each piece into a file
 - Need to replicate PK
- Combined
- Can speed up queries that require only on one partition, but queries that require data from all partitions may be slower

13

Another possibility to improve performance is to *replicate* a table, and store it twice in two different files; this can allow us to query them in parallel or at least reduce contention; however, we now have an integrity problem; when we modify the table, we need to modify it in both places, so this is better for tables that are not updated often.

Both partitioning and replication are much more useful when we're dealing with *distributed* databases, where the information is stored in more than one server. Distributed databases are outside the scope of this introduction, but are a really interesting topic.

One of the most important performance decisions is which fields and tables to use for indexes.

Indexes are secondary structures, that is, they do not actually contain any data at the logical level, but they help organize and search the data faster. There are two main kinds of indices, those based on search trees, or B-trees, and those based on hash tables; there are many other index structures for specialized applications.

When creating an index, we specify a field or expression in a field to be the *key* for the index; searches on that key will be sped up by the index, but searches on other keys or conditions will not be affected by the index.

Many times, on an index, we associate the key not with the actual data on a record, but with some sort of internal *pointer* to the record; in paged files, this pointer would consist of an identifier for the page in the file, and an offset within the page that corresponds to the actual record.

Replication

SPSU

- Storing pieces of data twice
- Improves performance by minimizing contention
- Integrity problem: Duplication
- Best for data not updated often

14

Indexes

SPSU

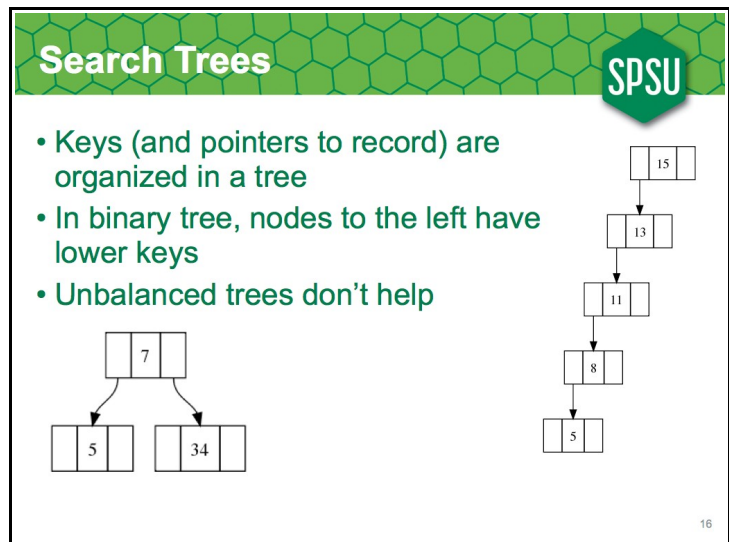
- Secondary data structures that improve searches
- Two main kinds
 - Search trees (and B-Trees)
 - Hash tables
- Some field or expression is the **key** for the structure
- Simplest would be to just have a list with the key and a pointer to the record
- Pointers to records would have page+offset

15

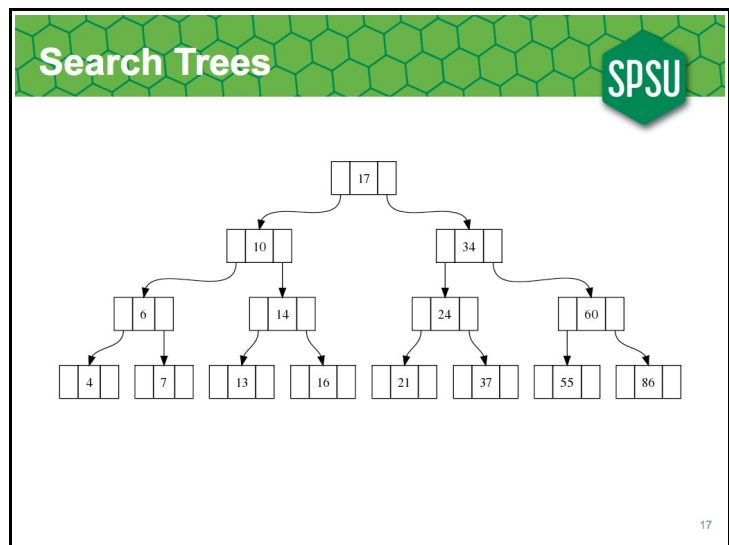
Many indexing techniques are based on search trees, and the simplest case is a binary search tree.

A binary search tree is composed of nodes, starting with a node called the *root node*. Each node contains a *key*, one *value* (which might be a pointer to actual data) and two pointers to the node's *children* (which may be pointing to a special empty location); this pointers are identified as *left* or *right* pointers, and the node's pointed to as the left or right children of the parent node.

In the left side of our slide, we see one node, with a key of 7 (the actual value is not shown) and pointers to its left and right children. The left child has a key of 5 and no children of its own, and the right child has a key of 34 and no children of its own. The figure on the right show an *unbalanced* search tree, which is not very useful, as we will see next.



The nodes in a binary search tree are organized so that all nodes to the left of a given node (that is, the node's left child, and all the descendants of that child) have key values that are lower than the parent node, and all of the nodes to the right have key values that are higher than the parent node, as illustrated in this slide; this makes searching easy and efficient; to find a given key, we start at the root node; if the search key is smaller than the node's key, then go to the node's left children; otherwise, go to its right children, and then continue in this fashion until we get to the bottom of the tree or we find the key.



If we have a *balanced* tree, that is, one where all the leaves are at about the same height, this makes searching very efficient; to simplify, let's assume that only the data on the leaves (that is, at the bottom of the tree) is important, and the keys in intermediate levels are just used to guide the search; then we see that we have 8 nodes at the bottom; to find a given one, we need to traverse only four nodes; starting and the root and following a path to that node; if we add one more level, we have 16 nodes and 5 comparisons; then 32 nodes and 6 comparisons, 64 and 7, and so on; we are doubling the number of nodes, but just doing one more comparison, which gives us logarithmic growth; we need 11 comparisons for 1024 nodes, only 21 for about a million nodes, and only 31 for about a billion nodes.

Summarizing, balanced search trees make searches way faster; logarithmically faster;

When dealing with databases, we are trying to minimize the number of pages read from disk, so we would normally want a node in our index to correspond to a page in disk; this means we want more than two children per node, which is called a *multiway* tree.

Tree based indices make searches for *their* key much faster than looking at the table; however, when we change data in the table, the index needs to be maintained, which adds some time to those updates.

Another indexing technique is the hash table; here the values are put into *buckets*, and the key determines which bucket they go into; to find the records for a given key, just apply a function to the key, and then go to that bucket. Normally, we want many different keys to map into the same bucket, to save on space; however, if we have too many *collisions*, then some buckets will *overflow*; there are many techniques for dealing with overflow, but some of the more powerful ones involve basically creating a new hash table for the records on the overflowing bucket, which essentially converts the hash table into a tree of hash tables, similar to the B-trees we have discussed.

Another issue is the design of the hash function; we want a function that distributes keys as evenly as possible among the buckets; however, this kind of functions end up destroying the ordering of the keys, so hash tables are more useful for *point* queries, where we are looking for a specific key, instead of *range* queries, where we look for a whole range of keys; search trees are useful for both kinds of queries.

Search Trees

SPSU

- Balanced trees make searches way faster
- Logarithmic instead of linear
 - 10 instead of 1,000
 - 20 instead of 1,000,000
- The base (of the logarithm) is not that important
- In DBs we use node size that fits in one page

18

Hash Tables

SPSU

- Place records (or key and pointer) on buckets
- Use a function on the key to find the appropriate bucket (which function?)
- Problems
 - collisions (several keys map to same bucket)
 - overflow (too many keys for the bucket)
- One good solution: Expand bucket to become another hash table ... so it is a tree !!
- Most good functions help only with point queries

19

Summarizing, at the SQL level we create indices with the CREATE INDEX command; indices make searches on their key much faster, but they make updates to the table slower, since the index needs to be updated.

Also, since indices reduce the search complexity from linear to logarithmic, they are more useful for large tables; with tables of 50 records or less, indices do not significantly speed up searches.

Since indices only speed up searches for their keys, and may slow down updates, we need to decide which fields are we going to create indices for; a rule of thumb would be to create indices for the primary keys of tables, for fields which are frequently searched, and for keys which frequently appear in GROUP BY expressions.

We can organize the records in a file in a number of different ways; we call these ways *file organizations* and we will briefly describe some of them in the next slide.

The main factors when deciding which organization to use for a given file is the speed of retrieval, how much space they use, and the need to avoid reorganization.

The simplest way would be to just write the records on pages as the need arise, with no ordering imposed. We can also organize the records sequentially, or even sort them; the problem with keeping the records sorted is that when we insert a new record we need to resort the file, which can be very slow.

Another way would be to actually organize the records depending on an index structure, either a hash table or a B-tree; this would make accessing the records by that key very fast, but would essentially force the DB to always read that index to get to the data, even when it doesn't need to, which slows down the access when not searching by that key.

Indexes

SPSU

- **CREATE INDEX ... ON Table(field)**
 - Creates secondary index
- **Searches faster, Updates slower**
- **Only speed search on their index keys**
- **When to use:**
 - large tables (50+?)
 - unique index for PK
 - for frequently searched fields
 - for COUNT, GROUP etc fields

20

File Organization

SPSU

- **Techniques for physically arranging records in a file**
- **Main Factors**
 - Fast retrieval/throughput
 - Space
 - Minimize reorganization

21

File Organizations

SPSU

- **Unordered**
- **Sequential / Sorted**
 - Problem - reorganization
- **Primary Indexed**
 - Records are organized according to an index
 - May help with sequential scan (sorted)
 - 'Wastes' Space
- **Secondary index**
 - Create index for some keys (not affecting records)
- **Clustered**
 - Store several kinds of records on the same page

22

Regardless of the main organization of records in a file, we can always have secondary indices, which do not affect how the records are stored, but keep a list of keys and *pointers* to the actual records, helping with searches. Notice that when using secondary indices we first need to find the pointer, by searching the index, and then go to the actual file that contains the data.

Most of the time, we store only one kind of record in a page, to simplify the DBMSs job; however, in some cases we may want to store a main record with some records associated with it on the same page; for example, if we had a table for people, and another table for keeping their emails, we may want to store the email records for a person next to the actual record for that person; this makes retrieving the person together with its emails much faster, but slows down retrieving the other person's data only, since we would still be bringing the emails from disk.

SQL is a high-level language that allows us to easily express queries; however, the DBMS would still need to translate SQL queries into more basic operations; how it translates this queries is called *query optimization*;

Some of the basic operations would be a *point query*, that is finding records with specific vales in a key, *range queries*, which is finding all records with values within a given range.

Sometimes, the DBMS needs to go through a whole table, we call this a *full scan*; for GROUP BY or ORDER BY clauses, it may need to go through the table in a particular order, which is called a *sequential scan*.

Many SQL queries include JOINS, and we consider the JOIN a low level operation, since it combines features of both queries and scans.

You should be able to see how indices could help with point and range queries, sequential scans and joins; full scans are a last-resort operation, since going through all the records in a table is usually very slow.

Query optimization is the set of operations through which a DBMS transforms a SQL query into these basic operations; SQL is high-level and declarative; we tell the DBMS which records we want, but we do not explicitly tell it *how* to go about finding those records. The DBMS can do the operations in different order, or it can substitute one low-level operation for another.

Modern DBMSs usually generate all possible ways that they can execute the query, and then try to estimate the cost of each one; they then choose the one with the lowest cost. This is called *cost based query optimization*.

Kinds of operations (low level)

SPSU

- Point query
 - Find all records with a specific value on a field
- Range query
 - Find all records with any of a range of values on a given field
- Full Scan
 - Go through all rows in a table
- Sequential Scan
 - Go through all rows in a table in a certain order
- Joins

23

Query Optimization

SPSU

- SQL is declarative (doesn't say HOW to do the query)
- DBMS has choices
 - Reordering operations
 - Using Indexes or not
 - Joins - starting from which side
 - Different algorithms
- Modern DBMS use Cost-based optimizers
 - Generate many possibilities
 - Estimate the 'cost' of each possibility
 - Do the 'cheapest' one

24


Since most databases reside on disks, techniques for making disks faster or more reliable are very important; RAID is a family of techniques that combines several disks into a disk array that is faster, more reliable or both. There are several RAID levels, identified by numbers.

Basically, RAID increases reliability by introducing redundancy; if we copy the same information to two different disks, then even if one fails we can still access the data; notice this redundancy is at the hardware level; we are writing the same *page* to two different disks. Raid 1 is straight mirroring, every page is written to two different disks, greatly increasing reliability; however, we now need twice as many disks to store the same amount of information.

Performance is increased by increasing parallelism; if we read half the pages from one disk and the other half from another, we have basically doubled *throughput*; but not *latency*; although it still takes the same amount of time to actually reach a given page, we can now read about twice as many pages in the same unit of time. This interleaving of pages is called *striping* and is done in RAID level 0.

There are several other RAID levels, including RAID 0+1, which combines both mirroring and striping, and RAID level 5, which increases read performance and reliability, while using only one extra disk, instead of twice as many.

RAID



- Redundant Array of Inexpensive Disks
- Redundancy improves reliability
- Parallelism improves performance
- Levels
 - 0 - striping
 - 1 - mirroring
 - 5 parity

25