**Zusammenfassung der Themen der Bachelorarbeit**

* Optimierung von SQL-Befehlen (**Query-Processing**) [GM08], [SKS11], [SZT12], [Win11]
  + Einfluss von Operationen veranschaulichen
  + Einfluss von Datentypen
  + Abfragen mit
    - Subqueries
    - COUNT
    - GROUP BY
    - LIMIT
    - UNION
  + Join – Strategien
    - Nested Loop
    - Sort – Merge
    - Hash Join
* Verschiedene Datenbankobjekte werden analysiert
  + Verschiedene Index – Techniken
    - B\* – Tree
    - Hash
    - Bitmap
  + Vergleich Views und MVs
* Partitionierung, Replikation und Skalierbarkeit in relationalen Datenbanken
* Vergleich mit der spaltenorientierten Datenbank Apache Cassandra

**High Performance MySQL**

**Benchmarking**

* Uniquely convenient and effective for studying what happens when you give systems work to do
* Can help to observe the system’s behavior under load => learn which changes are important
* Goals: reduce bad behavior und test how the application currently performs or simulate a higher load in the future
* Different techniques:
  + Benchmark the application as a whole (full-stack)
    - Tests the entire application, including the web server, the application code, the network, and the database => in the end you care about the performance of the whole application
    - MySQL is not always the application bottleneck

=> can be hard to create and even harder to set up correctly

* + Isolate MySQL (single-component)
    - If you want to compare different schemas or queries
    - To analyse a specific problem in the application
    - Faster than longer benchmarks => more rapid to see results
* If you design the benchmark badly, you can end up making bad decisions, because the results don’t reflect reality
* Data itself and the dataset’s size both need to be realistic => if possible, use a snapshot of actual production data
* Setting up a realistic benchmark can be complicated and time-consuming => simulate the larger application’s data and workload if no prod data

**Measures**

* **Throughput** = number of transactions per unit of time => standardized, database vendors try to optimize it and usual unit of measurement is transactions per second (sometimes transactions per minute)
* **Response time** (latency) = measures the total time a task requires (depending of the application in µs, ms, seconds or minutes). From this you can derive aggregate response times, such as average, maximum, minimum, and percentiles. Maximum response time is rarely a useful metric (not repeatable as well) => common to use *percentile response times* (f. ex. if the 95th percentile response time is 5ms, you know that the task finishes in 5ms or less than 95% of the time
* **Concurrency** = concurrency on the web server doesn’t necessarily translate to the database server; the only thing it directly relates to is how much data your session storage mechanism must be able to handle. More accurate measurement of concurrency on the web server is how many simultaneous requests are running at any given time. Measure whether throughput drops or response times increase when the concurrency increases. A website with “50,000 users at a time” might require only 10 or 15 simultaneously running queries on the MySQL server!
* **Scalability** = useful for systems that need to maintain performance under changing workloads. An ideal system should get twice as much work done (throughput) when you double the amount of workers try to complete the task. Most systems are not linearly scalable, and exhibit diminishing returns and degraded performance as you vary the parameters.
* It’s best to benchmark whatever is important to the users of the application ((in)formal requirements about expected response times, concurrency, …). Then try to satisfy all of the requirements

**Problems and mistakes**

* Using only a subset of real data size
* Using incorrectly distributed data (in real world may be “hot spots”) => randomly generated data is almost always unrealistically distributed
* Using unrealistically distributed parameters
* Benchmarking a distributed application on a single server
* Failing to match real user behavior (just clicking on all the links without reading the page)
* Running identical queries in a loop => at some point will be fully or partially cached
* Failing to check for errors => always check errors logs after the benchmarks
* Ignoring the warmup – phase of the system (maybe also analyze how long it takes to warm up the caches)
* Benchmarking too quick => needs to last a while
* Run a series of short benchmarks, such as 60-second runs, and conclude something about the system’s performance from that

**How long should the benchmark last?**

* Is it important to run the benchmark a meaningful amount of time
* Most likely you are interested in the system’s steady-state performance => you need to observe the system in a steady state
* Can take a lot of time, especially on servers with a lot of data and memory
* If you don’t know how long to run the benchmark, just let the benchmark run forever and observe, until you’re satisfied that the system is starting to become stable
* If you don’t have the time to do the benchmarks right, any time you do spend is wasted
* It’s important to capture as much information about the system under test (SUT) as possible while the benchmark runs => good idea: benchmark directory with subdirectories for each run’s results
* Benchmarks must/should be repeatable
* For analyzing the reports you should use plotting, because otherwise you won’t see phenomes
* See different full-stack and single-componenttools (use ***sysbench*** and***gnuplot***)

Chapter 3: Profiling Server Performance => read and summarize

Test sysbench

**Chapter 4: Optimizing Schema and Data Types**

* Must design your schema for the specific queries that you will run => involves trade – offs,
  + A denormalized schema can be speed up some types of queries, but slow down others
  + Adding counter and summary tables is a great way to optimize queries, but I can lead higher maintaining costs

**Choosing optimal data types**

* Choosing the right data types is crucial to getting good performance
* **Smaller is better**
  + Try to use the smallest data type that can correctly store and represent your data
  + Smaller ones are typically faster, because the use less space on the disk, in memory and in the CPU cache => also require fewer CPU cycles to process
  + If in doubt use the smallest type and if it’s getting problematic later, you can just change the type easily
* **Simple is good**
  + Fewer CPU cycles are usually required to process the operations on simpler data types
  + Integer are cheaper to compare than characters, because character sets and collations (sorting rules) make character comparisons complicated
* **Avoid NULL if possible**
  + Lots of tables include nullable columns even when the application does not need to store NULL (the absence of a value) => it’s the default
  + It’s best to specify columns as NOT NULL unless you intend to store NULL in them
  + It’s harder for MYSQL to optimize queries that refer to nullable columns, because they make indexes, index statistics and value comparison more complicated => uses more storage space and requires special processing inside MYSQL
  + A **indexed** nullable column requires an extra byte per entry and can cause a fixed-size index (such as an index on a single integer column) to be converted to a variable-sized one
  + Performance improvement from changing NULL columns to NOT NULL is usually small, however if you intend to use indexes try to avoid them
* Steps to decide which data types to chose

1. **Determine what general class** of types is appropriate: numeric, string, temporal, …
2. **Choose the specific type**: many data types can store the same kind of data but vary in the range of the value they can store , the precision they permit, or the physical space (on disk and in memory) they require. Some also have special behaviors or properties

* *DATETIME* and *TIMESTAMP* can store the same kind of data (date and time) to a precision of one second. TIMESTAMP only uses half as much storage space, is time-zone aware and has special auto updating capabilities. But a much smaller range of allowable values and sometimes special capabilities can be a handicap
* MYSQL supports many **aliases** (f.ex. INTEGER, BOOL, NUMERIC) => can be confusing, but they don’t affect performance
  + If you create a table with an aliased data type and then examine *SHOW CREATE TABLE*, you’ll see that MYSQL reports the **base type**

**Whole Numbers**

* There are two kinds of numbers: whole numbers and real numbers (numbers with a fractional part)
* For storing whole numbers, use one of the integer types:
  + TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT
  + Require 8, 16, 24, 32, and 64 bits of storage space
  + Can store values from -2(**N**-1) to 2(**N**-1)-1, where N is the number of bits of storage
* Integer types can optionally have an ***UNSIGNED*** attribute, which disallows negative values and approximately doubles the upper limit of positive values you can store => for ***TINYINT*** [0; 255] instead of [-128; 127]
* Signed and unsigned types use the same amount of storage space and the same performance
* Integer *computations* generally use 64-bit BIGINT integers, even on 32-bit architectures (except for aggregate functions, that use DECIMAL or DOUBLE)
* MYSQL lets you specify the “width” for integer types, f.ex. INT(11) => does not restrict the legal range of values, but simplifies the display name, for storage and computational purposes, INT(1) is identical to INT(20)

**Real Numbers**

* Real numbers are numbers that have a fractional part
* You can also use DECIMAL to store integers that are so large they don’t fit in BIGINT
* FLOAT and DOUBLE types support approximate calculations with standard floating-point math
* DECIMAL type is to store exact fractional numbers
* Both floating-point and DECIMAL types let you specify a precision => for DECIMAL, you can specify the maximum allowed digits before and after the decimal point
* Floating-point math is significantly faster, because the CPU performs the computations natively, but there is a loss of precision
* MYSQL packs the digits into a binary string (nine digits per four bytes)
* ***DECIMAL (18, 9)*** stores nine digits on each side of the point and uses 9 bytes in total: four for the digits before the decimal point, one for the decimal point itself, and four for the digits after the decimal point
* DECIMAL can have up to 65 digits
* Floating-point types typically use less space than DECIMAL to store the same range of values
  + FLOAT column uses four bytes of storage
  + DOUBLE consumes eight bytes and has greater precision and larger range of values than FLOAT
* As with integers, you’re choosing only the storage type
  + MySQL uses DOUBLE for its internal calculations on floating-point types
* Only use DECIMAL for storing financial data => there you could also use BIGINT and avoid both the imprecision of floating-point storage and the cost of the precise DECIMAL math

**String Types**

* Since MySQL 4.1, each string column can have its own character set and set of sorting rules for that character set, or collation
* Two major string types are VARCHAR and CHAR, which store character values
* Hard to explain how these values are stored on the disk => implementations are storage engine – dependant
* VARCHAR stores variable-length character strings and is the most common string data type
  + Require less storage space than fixed-length types, because it uses only as much space as it needs (less space is used to store shorter values)
  + Varchar uses 1 or 2 extra bytes to record the value’s length (1 byte < 255 bytes)
  + However, because the rows are variable-length, they can grow when you update them, which can cause extra work => behavior if it no longer fits is engine-dependent
  + Worth using varchar when the maximum column length is much larger than the average length
* CHAR is fixed-length and MYSQL always allocates enough space for the specified number of characters
  + Useful if you want to store very short strings or if all the values are nearly the same length
  + CHAR is also better than varchar for data that’s changed frequently, because a fixed-length row is not prone to fragmentation
  + For very short columns CHAR is also more efficient than VARCHAR => CHAR(1) use only one byte, but VARCHAR(1) would use 2 bytes because of the length byte
* The sibling types for CHAR and VARCHAR are BINARY and VARBINARY, which store binary strings
* Binary strings are very similar to conventional strings, but they store bytes instead of characters and padding is also different (MySQL pads ***BINARY*** values with \0 (the zero byte) instead of spaces and doesn’t strip the pad value on retrieval)
* Is useful if you want to store binary data and want to compare the values as bytes instead of characters
  + Advantage of byte-wise comparisons is more than just a matter of case insensitivity
  + Binary comparisons can be way simpler than character ones (also faster)
* Storing the value 'hello' requires the same amount of space in a VARCHAR(5) and a VARCHAR(200) column, but it is bad for sorting or operations that use in-memory temporary tables => allocate only as much space as you need
* Be careful with the BINARY type if the value must remain unchanged after retrieval. MySQL will pad it to the required length with \0s

**BLOB and TEXT types**

* Are string data types designed to store large amounts of data as either binary or character string respectively
* They are each families of data types:
  + Character types are TINYTEXT, SMALL TEXT, TEXT, MEDIUMTEXT, and LONGTEXT => BLOB is a synonym for SMALLBLOB
  + Binary types are TINYBLOB, SMALLBLOB, BLOB, MEDIUMBLOB, and LONGBLOB => TEXT is a synonym for SMALLTEXT
* Unlike with all other data types, MySQL handles each BLOB and TEXT value as an object with its own identity
* Only difference between the two is that BLOB types store binary data with no collation or character set, but TEXT types have a character set and collation
* Instead of sorting the full length of the string, it sorts only the first ***max\_sorth\_length*** of such columns
  + If you want to sort by only the first few characters, you can either decrease the ***max\_sorth\_length*** server variable or use ORDER BY SUBSTRING(*column, length*)
* Queries that use BLOB or TEXT columns and need an implicit temporary table => can result in a serious performance overhead
* Best solution is to avoid using the BLOB and TEXT types unless you really need them
  + If you can’t avoid them, you may be able to use the SUBSTRING(*column, length*) trick everywhere a BLOB column is mentioned => converts the values to character strings, which will permit in-memory temporary tables
  + Must be sure to use a short enough substring, so that the temporary table doesn’t grow larger than ***max\_heap\_table\_size*** or ***tmp\_table\_size***

**ENUM**

* Sometimes you can use a ENUM type instead of conventional string types
* An enum column can store a predefined set of distinct values
* MYSQL stores them very compactly, packed into one or two bytes depending on the number of values in the list
* Stores each value internally as an integer representing its position in the field definition list
* Keeps the “lookup table” that defines the number-to-string correspondence in the table’s ’.frm’ file
* This duality can be terribly confusing if you specify numbers for your ENUM constants => f.ex. ENUM('1', '2', '3') => We suggest you don’t do this
* ENUM field sorts by the internal integer values, not by the strings themselves
* Specify ENUM members in the order in which you want them to sort
* Or Use FIELD() to specify a sort order explicitly in your queries, but this prevents MySQL from using the index for sorting
* Biggest downside of ENUM is that the list of strings is fixed, and adding or removing strings requires the use of ALTER TABLE
* Can be slower to join a CHAR or VARCHAR column to an ENUM column than to another CHAR or VARCHAR column
* Example analysis with 2 columns converted to enum instead of varchar
  + Join is faster after converting the columns to ENUM, but joining the ENUM columns to VARCHAR columns is slower
  + Looks like a good idea, as long as they don’t have to be joined to VARCHAR columns
  + It’s a common design practice to use “lookup tables” with integer primary keys to avoid using character-based values in joins
  + Converting these two columns to ENUM made the table about 1/3 smaller according to SHOW TABLE STATUS => in some cases beneficial even if the ENUM columns need to join to VARCHAR columns
  + Primary key itself is only about half the size after the conversion => also good for the indexes

**Date and time types**

* MYSQL has many types, f.ex. such as YEAR and DATE
* Finest granularity of time MYSQL can store is one second
* However, it can do temporal computations with microsecond granularity
* Most of the temporal types have no alternatives, so there is no question of which is the best choice => only question is what to do when so store oth the date and the time: ***DATE***, ***TIME***, ***TIMESTAMP***
* **DATETIME** = can hold large range of values from the year 1001 to the year 9999 with a precision of one second
  + Stores the date and time packed into an integer in YYYYMMDDHHMMSS format, independent of the time zone
  + Uses 8 byte of storage space
  + By default MYSQL displays DATETIME values in a sortable, unambiguous format, like this 2008-01-16 22:37:08
* **TIMESTAMP** = stores the number of seconds elapsed since midnight, January 1, 1970 (GMT)
  + Uses only four bytes of storage, so it has a smaller range from 1970 partway through 2038
  + Value of timestamp displays also depends on the time zone
  + Has special properties that DATETIME doesn’t have => changes from each version
    - By default MYSQL will set the first TIMESTAMP column to the current time when you insert a new row => same for update
    - Can configure the insertion and update behaviors for any TIMESTAMP column
  + TIMESTAMP columns are NOT NULL by default
* If you can use TIMESTAMP, you should use it, because it is more space-efficient
* For storing microseconds, you need use BIGINT or DOUBLE
* ***BIT*** type should be avoided to store one or many true/false values in a single column
* If you need to store many true/false values, consider combining many columns into one with MySQL’s native ***SET*** data type => change of the column’s definition is only possible with an ALTER TABLE (very expensive and you can’t use indexes)
* Bitwise operations on integer columns: better to change the “enumeration” but need a good understanding of bitwise manipulations

**Choosing identifiers**

* Choosing a good data type for an identifier column is very important, because you are more likely to compare these columns to other values, use them for lookups or more likely to use them as foreign keys in other tables
* Need to consider not only the storage type, but also how MYSQL performs computations and comparisons on that type
* F.ex. ENUM and SET types are internally integers, but are converted to strings when doing comparisons in a string context
* Once you chose a type, make sure to use the same type in all related tables => types should match exactly the same including properties like UNSIGNED
  + Otherwise, you would have performance issues or if it works, you can get hard-to-find errors
* Again: Choose the smallest size that can hold your required range of values and leave room for future growth if necessary
* ***INTERGER*** also usually the best choice for identifiers, because they’re fast and work with ***AUTO\_INCREMENT***
* Avoid Enum or Set types and also string types because of performance issues and space storage (in case of strings)
* Badly written schema migration programs and programs that autogenerate schemas can cause severe performance problems
* Applications that “hide complexity from developers” usually don’t scale well => think carefully before trading performance for developers’ productivity and test on a realistically large dataset, so you don’t discover performance issues to late
* MYSQL`s specific problems: too many columns, too many joins (as a rough rule of thumb, it’s better to have a dozen or fewer tables per query if you need queries to execute very fast with high concurrency), wrong use of enum, use NULL if you don’t know how to represent an unknown value (instead of inventing different methods to represent NULL, like ‘0000-00-00 00:00:00 ‘

**Normalization and denormalization**

* Usually, many different ways to represent any given data, ranging from fully normalized to fully denormalized and anything in between
* In a normalized database, each fact is represented once and only once
* In a denormalized database, information is duplicated, or stored in multiple places
* People with performance issues are advised to normalize their schemas, especially if the workload is write – heavy
* Positive reasons:
  + Normalized updates are usually faster than denormalized updates
  + When the data is well normalized, there’s less data to change
  + Normalized tables are usually smaller, so they fit better in memory and perform better
  + The lack of redundant data means there’s less need for ***DISTINCT*** or ***GROUP BY*** queries when retrieving lists of values
  + Consider the preceding example: it’s impossible to get a distinct list of departments from the denormalized schema without DISTINCT or GROUP BY, but if DEPARTMENT is a separate table, it’s a trivial query
* Any nontrivial query on a well-normalized schema will probably require at least one join, and perhaps several
* Not only expensive, but can also make some index strategies impossible => normalizing may place columns in different tables that would benefit from belonging to the same index
* Denormalized tables can have performance advantages, because less lookups and joins are required
* Fully (de-) normalized schemas are like laboratory rats => they have little to do with the real world
* Often you need to mix the approaches, possibly using a partially normalized schema, cache tables or other techniques
* Sometimes the best way to improve performance is to keep redundant data in the same table as the data from which it was derived
* Other times you need to build completely separate summary or cache tables, especially tuned for your retrieval needs
* “cache tables” = refer to tables that contain data that can be easily, if more slowly, retrieved from the schema (i.e., data that is logically redundant)
  + Cache tables are useful for optimizing search and retrieval queries
  + Might need many different index combinations to speed up various types of queries
  + Conflicting requirements sometimes demand that you create a cache table that contains only some of the columns from the main table
* “summary table” = tables that hold aggregated data from GROUP BY queries
  + F.ex. a count of the number of messages posted during the previous 24 hours, it would be useful to have a summary table, that is (re-) generated every hour => counts are not 100% accurate, but nearly, otherwise it is expensive to compute these statistics in real time
* When you rebuild summary and cache tables, you’ll often need their data to remain available during the operation => achievable by using a “shadow table”, which is a table you build “behind” the real table

**Materialized Views**

* Many database management systems offer these features, MYSQL doesn’t
* Are views that are actually precomputed and stored as tables on disk and can be refreshed or updated through various strategies
* FlexViews (special implementation of a developer) use deltas (difference between the before and after data), which is much more efficient than reading the data from the source table

**Counter Tables**

* An application that keeps counts in a table can run into concurrency problems when updating the counters
* Using a separate table can help you avoid query cache invalidations and lets you use some of the more advanced techniques
* Use a random slot variable for better concurrency (and use it together with the day as a primary key) => update the counter even ***ON DUPLICATE KEY UPDATE***
* Write a periodic job that merges all results into slot 0 and deletes every other slot

**Faster Reads, Slower Writes**

* Often need extra indexes, redundant fields, or even cache and summary tables to speed up read queries
* At the same time they add work to write queries and maintenance jobs
* For faster read queries you also increase development complexity for both read and write operations
* Still a technique that is commonly used if you design high performance

**Speeding Up ALTER TABLE**

* ALTER TABLE performance can become a problem with very large tables
* MYSQL performs most alterations like this:
  + Create a new empty table with the desired new table
  + Inserting all the data from the old table into the new one
  + Deleting the old table
* Can take a lot of time especially if you’re short on memory, the table is large and has a lot of indexes => can possibly take up some days
* MYSQL 5.1 or newer support for some types of “online” operations that won’t lock the table for the whole operation
* For general case either use operational tricks (like swapping servers around and performing the alter table on the not production service) or
* “shadow copy” – approach: build a new table with the desired structure beside the existing one, and then perform a rename and drop to swap the two
* Expensive way to change the default of a column: MODIFY COLUMN rental\_duration TINYINT(3) NOT NULL DEFAULT 5;
* Shorter and easier way: ALTER COLUMN rental\_duration SET DEFAULT 5;
  + Default value for the column is stored in the table’s ***.frm***file, so you should be able to change it without touching the table itself
* Potential operations without a table rebuild:
  + Remove (but not add) a column’s AUTO\_INCREMENT attribute
  + Add, remove, or change ENUM and SET constants
* Following technique:

1. Create an empty table with *exactly the same layout*, except for the desired modification (such as added ENUM constants)
2. Execute FLUSH TABLES WITH READ LOCK. This will close all tables in use and prevent any tables from being opened
3. Swap the ***.frm***files
4. Execute UNLOCK TABLES to release the read lock

* There’s also an example on how exactly this works (p. 143)

**Indexing for High Performance**

* Indexes = are data structures that storage engines use to find rows quickly
* Are critical for good performance and become more important as your data grows larger
* Lightly loaded databases often perform well even without proper indexes => performance can drop very quickly, as the dataset grows
* “Index optimization is perhaps the most powerful way to improve query performance” *=> very strong statement (discuss in BA)*
* Creating truly optimal indexes will often require you to rewrite the queries

**Indexing Basics**

* Easiest way to understand: Think about the index in a book. To find where a particular topic is discussed in a book, you look in the index, and it tells you the page number(s) where the term appears
* In MySQL a storage engine uses indexes in a similar way. It searches the index’s data structure for a value. When it finds a match, it can find the row that contains the match
* Following example: SELECT first\_name FROM sakila.actor WHERE ***actor\_id*** = 5;
  + There’s an index on the ***actor\_id*** column, so MySQL will use the index to find rows whose ***actor\_id*** is 5 => in other words: performs a lookup on the values in the index and returns any rows containing the specified value
* An index contains values from one or more columns in a table
* For more than one column, the column order is very important, because MySQL can only search efficiently on a leftmost prefix of the index
* Creating an index on two columns is not the same as creating two separate single-column indexes
* If you rely on an object-relational mapping (ORM) tool, you still need to learn about indexing
* There are many types of indexes, each designed to perform well for different purposes
* Indexes are implemented in the storage engine layer, not the server layer => that’s why they are not standardized (slightly different in every engine and not all engines support all types of indexes)

**B-Tree Indexes**

* Uses a B – Tree data structure to store it’s data
* Most of MySQL’s storage engines support this index type
* Use “B-Tree” for these indexes because that’s what MySQL uses in CREATE TABLE and other statements
* Different implementation and usage of B-Tree-Indexes depending on storage engine
* General idea of B-Tree is that all values are stored in order and each leaf page is the same distance from the root
* B-Tree index speeds up data access because the storage engine doesn’t have to scan the whole table to find the desired data
  + Instead, it starts at the root node (not shown in the graphic)
  + Slots in the root node hold pointers to child nodes and the storage engine follows these pointers
  + Finds the right pointer by looking at the values in the node pages, which define the upper and the lower bounds of the values in the child nodes
  + Ein Bild, das Text, Diagramm, Reihe, Screenshot enthält.

    Automatisch generierte BeschreibungEventually, the storage engine either determines that the desired value doesn’t exist or successfully reaches a leaf page
  + Leaf pages are special, because they have pointers to the indexed data instead of pointers to other pages
  + There might be many levels of node pages between the root and the leaves => the tree’s depth depends on the size of the table
* B-Trees store the indexed columns in order, so they’re useful for searching ranges of data
* F.ex. descending the tree for an index on a text field passes through values in alphabetical order => looking for every name that begins with “K” is efficient
* Index sorts the values according to the order of the columns given in the CREATE TABLE statement, like ***primary key(last\_name, first\_name, dob)***
* B-Tree-indixes work well for lockups by the full key value, a key range or a key prefix => useful only if the lookup uses a leftmost prefix of the index
* Following kinds of queries are useful:
  + Match the ***full value***: match on the full key value specifies values for all columns in the index, f.ex. Cuba Allen who was born on 1960-01-01
  + Match a ***leftmost prefix***: uses only the first column in the index, f.ex. Allen
  + Match a ***column prefix***: first part of a column’s value, f.ex. last names that begin with ‘M’
  + Match a ***range of values***, f.ex. find people whose last names are between Allen and Barrymore
  + Match ***one part exactly and match a range on another part***: exact match on last\_ name and a range query on first\_name, f.ex. last name is Allen and whose first name starts with the letter ‘K’
  + ***Index only queries***: are queries that access only the index, not the row storage
* Because tree nodes are sorted, they can be used for both lookups (finding values) and ORDER BY queries (finding values in sorted order)
* If a B-Tree can help you find a row in a particular way, it can help you sort rows by the same criteria
* **Limitations of B-Tree indexes**
  + Not useful if the lookup does not start from the leftmost side of the indexed columns, f.ex. only match the first name or the date and not possible to match the end of the last name
  + You can’t skip columns in the index, specific LAST\_NAME and specific DATE won’t work, unless you specify a specific FIRST\_NAME
  + Storage engine can’t optimize accesses with any columns to the right of the first range condition
    - WHERE last\_name="Smith" AND first\_name LIKE 'J%' AND dob='1976-12-23'
    - Index access will use only the first two columns in the index, because the LIKE is a range condition
* For optimal performance => create indexes with the same columns in different orders to satisfy your queries

**Hash indexes**

* Hash index is built on a hash table and is useful only for the exact lookups that use every column in the index
  + For each row the storage engine computes a ***hash code*** of the indexed columns
  + Hash code = a small value that will probably differ from the hash codes computed for other rows with different key values
* In MySQL only the Memory storage engine supports explicit hash indexes
* Default index type for Memory tables, but B-Tree is possible as well
* Memory engine supports nonunique hash indexes
  + If multiple values have the same hash code, the index will store their row pointers in the same hash table entry, using a linked list
* Example: ***SELECT lname FROM testhash WHERE fname='Peter'***
  + MySQL will calculate the hash of 'Peter' and use that to look up the pointer in the index
  + f('Peter') = 8784 => MySQL will look in the index of 8784 and find the pointer to row 3
  + Final step is to compare the value in row 3 to 'Peter', to make sure it’s the right row
* Indexes themselves store only short hash values => hash indexes are very compact, and lookups are lightning fast

**Limitations**

* Index only contains hash codes and row pointers rather than the values themselves, MySQL can’t use the values in the index to avoid reading the rows => accessing the in-memory rows is very fast, so this doesn’t degrade performance
* Can’t use hash indexes for sorting => aren’t stored in a sorted order
* Don’t support partial key matching, because they compute the hash from the entire indexed value => if you have ***(A, B)*** and the ***WHERE*** - clause only refers to ‘A’ the hash index won’t help
* Hash indexes support only equality comparisons like ***‘=’***, ***‘<=>’*** or ***IN()*** – operators => no speed up of range queries
* Accessing data in a hash index is very fast, unless there are many collisions (multiple values with the same hash) => storage engine must follow each row pointer in the linked list and compare their values to the lookup value to find in the right row(s)
* Some index maintenance operations can be slow if there are many hash collisions, f.ex. if you create an index on a column with a very low selectivity (many hash collisions) and the delete a row from the table => finding the pointer from the index might be expensive
* All these limitations make hash indexes useful only in special cases
* When they meet the application’s needs, they can improve performance dramatically => f.ex. in DWH with the star-schema that requires lots of joins
* Some storage engines (InnoDB) can notice that some index values are being accessed very frequently and it builds a hash index for them in memory on top of B-Tree indexes

**Building your own hash indexes**

* You can emulate hash indexes => gives you access to some of the desirable properties of hash indexes, such as very small index size for very long keys
* Idea

1. Create a pseudo hash index on top of a standard B-Tree index
2. Will still use the B-Tree index for lookups, but it will use the keys’ hash values for lookups, instead of the keys themselves
3. Specify the hash function manually in the query’s ***WHERE*** clause

* Works well for URL lookups => ***SELECT id FROM url WHERE url="http://www.mysql.com"*** *->* ***AND url\_crc=CRC32("http://www.mysql.com");***
* Works well because the MySQL query optimizer notices there’s a small, highly selective index on the ***url\_crc*** column
* Faster than the alternative to index the full URL as a string
* Negative: the need to maintain the hash values
* Create table with ***url*** and ***url\_crc*** column and ***trigger*** on insert or update
* Don’t use complex hash functions with long (string) returns, because it’s a waste in storage space and performance => **simple hash functions** can offer acceptable collision rates with better performance

**Handling hash collisions**

* When you search for a value by its hash, you must also include the literal value in your WHERE clause => otherwise it would return more than 1 value if there’s an hash collision
* Hash collisions grows faster than you might think => so-called Birthday Paradox
* RC32() returns a 32-bit integer value, so the probability of a collision reaches 1% with as few as 93,000 values
* If collisions aren’t a problem, f.ex. because of statistical queries, you can simplify and gain some efficiency, by using less bit integer values

**Spatial (R-Tree) indexes**

* Used with partial types such as GEOMETRY
* Don’t require your WHERE clauses to operate on a leftmost prefix of the index
* Index the data by all dimensions at the same time
* Lookups can use any combination of dimensions efficiently
* Must use the MySQL GIS functions, such as MBRCONTAINS(), for this to work => MySQL’s GIS support isn’t great (most don’t use it)
* Go-to solution for GIS in an open source RDBMS is PostGIS in PostgreSQL

**Full-text indexes**

* Special type of index that finds keywords in the text instead of comparing values directly to the values in the index
* Full-text searching is completely different from other types of matching
* Has many subtleties, such as stopwords, stemming and plurals, and Boolean searching
* Much more analogous to what a search engine does than to simple WHERE parameter matching
* Having a full-text index on a column does not eliminate the value of a B-Tree index on the same column
* Full-text indexes are for MATCH AGAINST operations, not ordinary WHERE clause operations

**Other types of index**

* Several third-party storage engines use different types of data structures for their indexes
* TokuDB uses fractal tree indexes => newly developed data structure that has some of the same benefits as B-Tree indexes, without some of the drawbacks
* InnoDB includes clustered indexes and covering indexes
* ScaleDB uses Patricia tries, and other technologies such as InfiniDB or Infobright have their own special data structures for optimizing queries

**Benefits of Indexes**

* Enables the server to navigate quickly to a desired position in the table => have several additional benefits, based on the properties of the data structures used to create them
* For B-Tree indexes MySQL can exploit the sorted data for queries with clauses such as ORDER BY and GROUP BY
* Because the data is presorted, a B-Tree index also stores related values close together
* Index actually stores a copy of the values, so some queries can be satisfied from the index alone, three main benefits:
  + Indexes reduce the amount of the data the server must examine
  + Indexes help the server to avoid sorting and temporary tables
  + Indexes turn random I/O to sequential I/O
* Read this **“*Relational Database Index Design and the Optimizers*, by Tapio Lah- denmaki and Mike Leach (Wiley)”** for more information

**Is an Index the best solution?**

* Isn’t always the right tool
* Index = most effective when they help the storage engine find rows **without** adding more work than they avoid
* For small tables it is often more effective to simply read all the rows in the table
* For medium to large tables, indexes can be very effective
* For enormous tables, the overhead of indexing, as well as the work required to actually use the indexes, can start to add up => need to choose a technique that identifies groups of rows **(partitioning)**
* For lots of tables it can make sense to create a metadata table to store some characteristics of interest for your tables (partitioning into multiple tables)
* At the scale of terabytes, locating individual rows doesn’t make sense; indexes are replaced by per-block metadata

**Indexing Strategies for High Performance**

* There are many special-case optimizations and specialized behaviors

**Strategy 1: Isolating the column**

* MySQL generally can’t use indexes on columns unless the columns are isolated in the query
* “Isolating” the column means it should not be part of an expression or be inside a function in the query, not like this: ***WHERE actor\_id + 1 = 5;***
* Human could easily simply it, but MySQL can’t => get in the habit of simplifying your WHERE criteria, so that the indexed column is alone on one side of the comparison operator

**Strategy 2: Prefix Indexes and Index Selectivity**

* Sometimes need to index very long character columns => makes the indexes large and slow
* One strategy is to use a hash index as explained
* Can often save space and get good performance by indexing the **first few characters** instead of the whole value => less space, but also less selective
* Index selectivity = ratio of the number of **distinct indexed values** (the *cardinality*) to the total **number of rows** in the table (*#T*), and ranges from **1/*#T***to **1**
* Highly selective index is good because it lets MySQL filter out more rows when it looks for matches => unique index has a selectivity of 1
* A **prefix** of the columns is often selective enough to give good performance
* For ***BLOB*** or ***TEXT*** or very long ***VARCHAR***, you must define prefix indexes
* Choose a prefix that’s long enough to give **good selectivity**, but **short enough** to save space => you’d like the prefix’s cardinality to be close to the full column’s cardinality
* MySQL cannot use prefix indexes for ***ORDER BY*** or ***GROUP BY*** queries, nor can it use them as covering indexes
* Long hexadecimal identifiers benefit especially from prefix indexes
* Sometimes suffix indexes make sense (e.g., for finding all email ad- dresses from a certain domain) => MySQL does not support it natively, but you can store a reversed string and index a prefix of it (+ use triggers to maintain the index)

**Multicolumn Indexes**

* Lahdenmaki and Leach’s book introduces a three-star system for grading how suitable an index is for a query
  + Index earns one star if it places relevant rows adjacent to each other
  + A second star if its rows are sorted in the order the query needs
  + A final star if it contains all the columns needed for the query
* Common mistakes are to index many or all the columns separately, or to index columns in the wrong order
* First mistake is to index many columns separately => often results when people give vague but authoritative-sounding advice such as “create indexes on columns that appear in the WHERE clause”
* Only result in one-star indexes at best
* Sometimes if you can’t design a three-star index, it’s much better to ignore the WHERE clause and pay attention to optimal row order or create a covering index instead
* MySQL can cope a little with such poorly indexed tables by using a strategy known as ***index merge***, which permits a query to make limited use of multiple indexes from a single table to locate desired rows
* Example
  + Table film\_actor with index on ***film\_id*** and ***actor\_id***
  + Neither is a good choice for both WHERE conditions
  + In older MySQL it would result in a table scan unless you wrote it as the UNION of two queries
  + In MySQL 5.0 or newer the query can use both indexes, scanning them simultaneously and merging the results
  + Three variations on the algorithm:
    - Union for OR conditions
    - Intersection for AND conditions
    - Unions of intersections for combinations of the two
* When the server intersects indexes (usually for ***AND*** conditions), it usually means that you need a single index with all the relevant columns, not multiple indexes that need to be combined
* When the server unions indexes (usually for ***OR*** conditions), sometimes the algorithm’s buffering, sorting, and merging operations use lots of CPU and memory resources => especially true if not all the indexes are very selective, so the scans return lots of rows of the merge operation
* **Optimizer** doesn’t account for this cost => optimizes just the number of random page reads => can make it “underprice” the query, so that it is slower than a full table scan
* Sometimes rewriting such queries with a ***UNION***, the way you used to have to do in MySQL 4.1 and earlier, is more optimal
* You can disable index merges with the ***optimizer\_switch*** option or variable or also use ***IGNORE INDEX***

**Choosing a good column order**

* One of the most common causes of confusion
* Correct order depends on the queries that use the index
* Choose the index order such that rows are sorted and grouped in a way that will benefit the query
* Only applies to B-Tree Indexes => Hash and other index types don’t store their data in sorted order
* Multicolumn B-Tree index means that the index is sorted first by the leftmost column => index can be scanned in either forward or reverse order to satisfy queries like *ORDER BY,* etc.
* Column order either enables or prevents the index from earning “stars” in Lahdenmaki and Leach’s three-star system
* Place the most **selective columns** first in the index => can be good when there is no sorting or grouping to consider => purpose is only to optimize ***WHERE*** lockups
* Depends also on the **actual values** you use to look up rows (=> the distribution of values)
* Choose the column order such that it’s as selective as possible for the queries that **you’ll run most**
* Choose the column order such that it’s as selective as possible for the queries that you’ll run most

**Clustered Indexes**

* Not a separate type of index, rather they’re an **approach to data storage**
* InnoDB’s clustered indexes store a B-Tree index and the rows together in the **same structure**
* “Clustered” refers to the fact that rows with adjacent key values are stored close to each other
* When a table has a clustered index, its rows are stored in the index’s leaf pages => can only have 1 clustered index per table (can’t store rows in two places at once)
* However, *covering indexes* let you emulate multiple clustered indexes
* Not all storage engines support clustered indexes
* **Leaf pages** contain full rows => **node pages** contain only the indexed columns
* Some database servers let you choose which index to cluster, some just use the primary key (InnoDB even defines a hidden PK if no unique nonnullable index)
* Clustering PKs can help performance, but it can also cause serious performance problems
* Advantages of clustering data
  + Keep related data together
  + Data access is fast => retrieving data faster than lockups in a nonclustered index
  + Queries that use covering indexes can use the PK values contained in the leaf node
* Disadvantages
  + Clustering gives the largest improvements for I/O-bound workloads => if the data fits in memory the order doesn’t really matter => no benefits
  + Insert speed depend heavily on insertion order => inserting rows in PK order is the fastest way to load data into the table
  + Might be a good idea to reorganize the table with ***OPTIMIZE TABLE*** after loading a lot of data, if you didn’t load the rows in PK order
  + Updating the clustered index is expensive => forces to move each updated row to a new location
  + Tables are subject to ***page splits***when new rows are inserted => must split the page into two to accommodate the row, can cause a table to use more space on disk
  + Clustered tables can be slower for full table scans, especially if rows are less densely packed or stored nonsequentially because of page splits
  + Secondary (nonclustered) indexes can be larger than you might expect, because their leaf nodes contain the primary key columns of the referenced rows
  + Secondary index accesses require two index lockups instead of one => because a leaf node doesn’t store a pointer to the referenced row’s physical location; rather, it stores the row’s primary key values

**Comparison of InnoDB (clustered) and MyISAM (nonclustered) layout**

* Suppose the table is populated with **primary key** values 1 to 10,000, inserted in random order and then optimized with ***OPTIMIZE TABLE*** => data optimally arranged on disk but the rows might be in a random order
* Values for ***col2*** are randomly assigned between 1 and 100 (a lot of duplicates)

MyISAM’s data layout

* Stores the rows on disk in the order in which they were inserted
* Rows are **fixed-size**, MyISAM can find any row by seeking the required number of bytes from the beginning of the table
* Each leaf in the index can simply contain the row number of the **layout table**
* ***Col2*** is just an index like any other
* No structural difference between a primary key or any other index => PK is simply a unique, nonnullable index named PRIMARY

InnoDB’s data layout

* Stores the same data very differently because of its clustered organization
* The clustered index “is” the table in InnoDB, there’s no separate row storage as there is for MyISAM
* **Each leaf node** in the clustered index contains the primary key value, the transaction ID, and rollback pointer InnoDB uses for transactional and MVCC purposes, and the **rest of the columns** (in this case, col2)
* Instead of storing “row pointers,” InnoDB’s secondary index leaf nodes contain the primary key values, which serve as the “pointers” to the rows => reduces the work needed to maintain secondary indexes, but makes it the index larger
* InnoDB’s non-leaf B-Tree nodes each contain the indexed column(s), plus a pointer to the next-deeper node (which might be either an- other non-leaf node or a leaf node)

**When Primary Key Order Is Worse**

* For high-concurrency workloads, inserting in primary key order can create points of contention in InnoDB
* Upper end of the primary key is one hot spot => all inserts take place there, concurrent inserts might fight over next-key locks
* Another hot spot is AUTO\_INCREMENT locking mechanism
* If you experience problems with that, you might be able to redesign your table or application, or configure **innodb\_autoinc\_lock\_mode**

**Inserting rows in primary key order with InnoDB**

**Covering indexes**

**Using Index Scans for Sorts**

**Packed (Prefix-Compressed) Indexes**

**Redundant and Duplicate Indexes**

* MySQL allows you to create multiple indexes in the same column
* MySQL must maintain each duplicate index separately and the query optimizer will consider each of them when it optimizes queries => can impact performance and does not “notice” and protect you from your mistake
* Duplicate indexes are indexes of the same type, created on the same set of columns in the same order => try to avoid creating / remove them