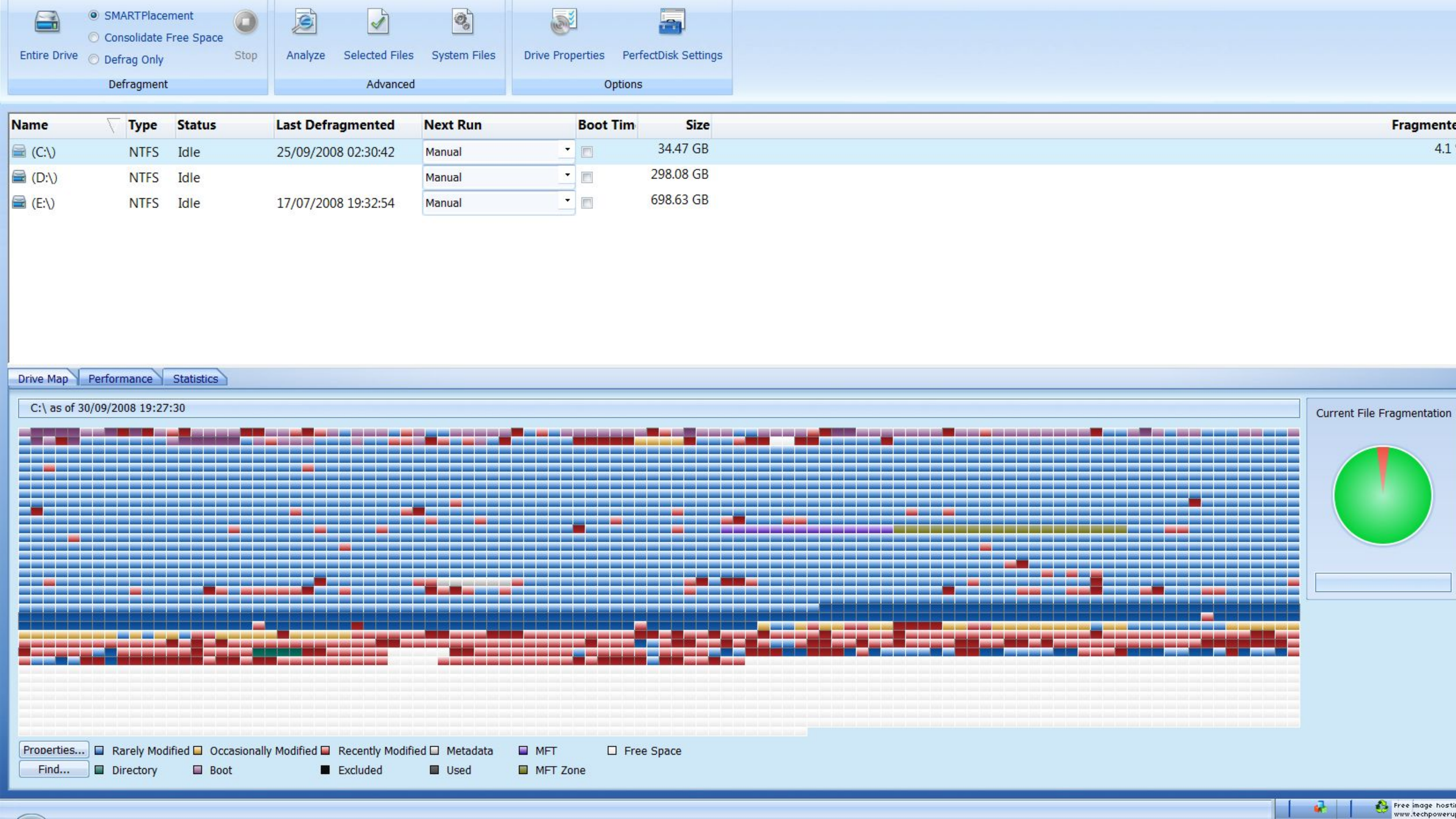
Database Management Systems

INFSCI 1022

Indexes

Functions



[✓ SMARTPlacement] Consolidate Free Space Defrag Only

Name	Type	Status	Last Defragmented	Next Run	Boot Tim	Size	Fragmente
(C:\)	NTFS	Idle	25/09/2008 02:30:42	Manual	☐	34.47 GB	4.1
(D:\)	NTFS	Idle		Manual	☐	298.08 GB	
(E:\)	NTFS	Idle	17/07/2008 19:32:54	Manual	☐	698.63 GB	

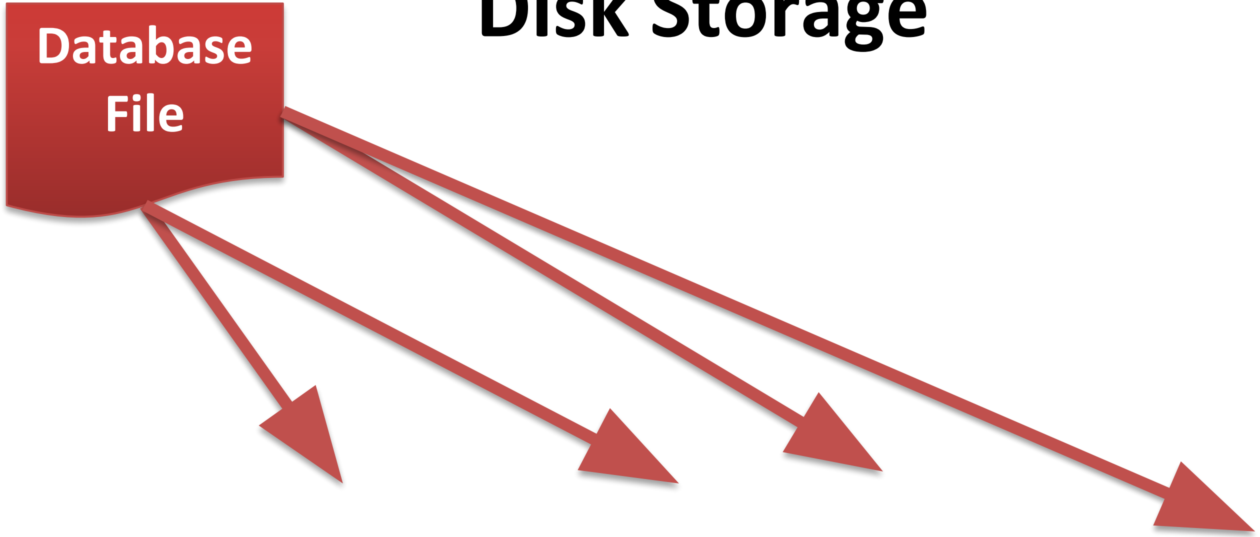
C:\ as of 30/09/2008 19:27:30

🟦 Rarely Modified	🟨 Occasionally Modified	🟥 Recently Modified	☐ Metadata	🟪 MFT	☐ Free Space
🟩 Directory	🟪 Boot	⬛ Excluded	⬛ Used	🟫 MFT Zone	

Disk Storage

- Data on disks (hard drives) is stored in blocks
- These blocks are accessed in their entirety (atomic disk access operation)
- A single file is not necessarily stored in a single block (a large file could be split across multiple blocks)

Disk Storage



- Disk blocks contain a section for data, a pointer to the location of the next node (or block)

Linear Search

```
SELECT * FROM
employees.departments
WHERE dept_no = ‘d006’;
```

dept_no	**dept_name**
d009	Customer Service
d011	Department 1
d012	Department 2
d005	Development
d002	Finance
d003	Human Resources
d001	Marketing
d004	Production
d006	Quality Management
d008	Research
d007	Sales
d010	Software Engineering

Linear Search

- Searching on a field that isn’t sorted requires a **Linear Search**
- **Linear Search** requires N/2 block accesses (on average), where N is the number of blocks that the table spans.
- If that field is a non-key field (i.e. doesn’t contain unique entries) then the entire table space must be searched at N block accesses.

Binary Search

```
SELECT * FROM
employees.departments
WHERE dept_no = ‘d006’;
```

dept_no	**dept_name**
d001	Marketing
d002	Finance
d003	Human Resources
d004	Production
d005	Development
d006	Quality Management
d007	Sales
d008	Research
d009	Customer Service
d010	Software Engineering
d011	Department 1
d012	Department 2

Binary Search

- Binary search (half-interval search) algorithm finds the position of a specified input value (the search "key") within a list sorted by key value.
- For binary search, the list should be arranged in ascending or descending order.
- In each step, the algorithm compares the search key value with the key value of the middle element of the list.

Binary Search

- If the keys match, then a matching element has been found and its index, or position, is returned.
- If the search key is less than the middle element's key, then the algorithm repeats its action on the sub-list to the left of the middle element.
- If the search key is greater than the middle element's key, then the algorithm repeats its action on the sub-list to the right of the middle element.
- If the remaining list to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

Binary Search Example

- Task: Find the index (location) of **d006**

d001 d002 d003 d004 d005 d006 d007 d008 d009 d010 d011 d012 d013

Binary Search Example

- Step 1: Split the list in half, find the middle element

d001 d002 d003 d004 d005 d006 **d007** d008 d009 d010 d011 d012 d013

Middle element

Binary Search Example

- Step 2: Is d006 less than or greater than d007?

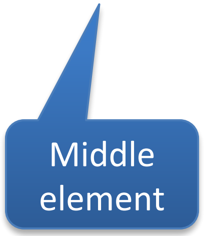
d001 d002 d003 d004 d005 d006 **d007** d008 d009 d010 d011 d012 d013

Middle element

Binary Search Example

- Step 3: Since d006 is less than, discard the right side of the list. Repeat the algorithm until a match is found

d001 d002 d003 **d004** d005 d006



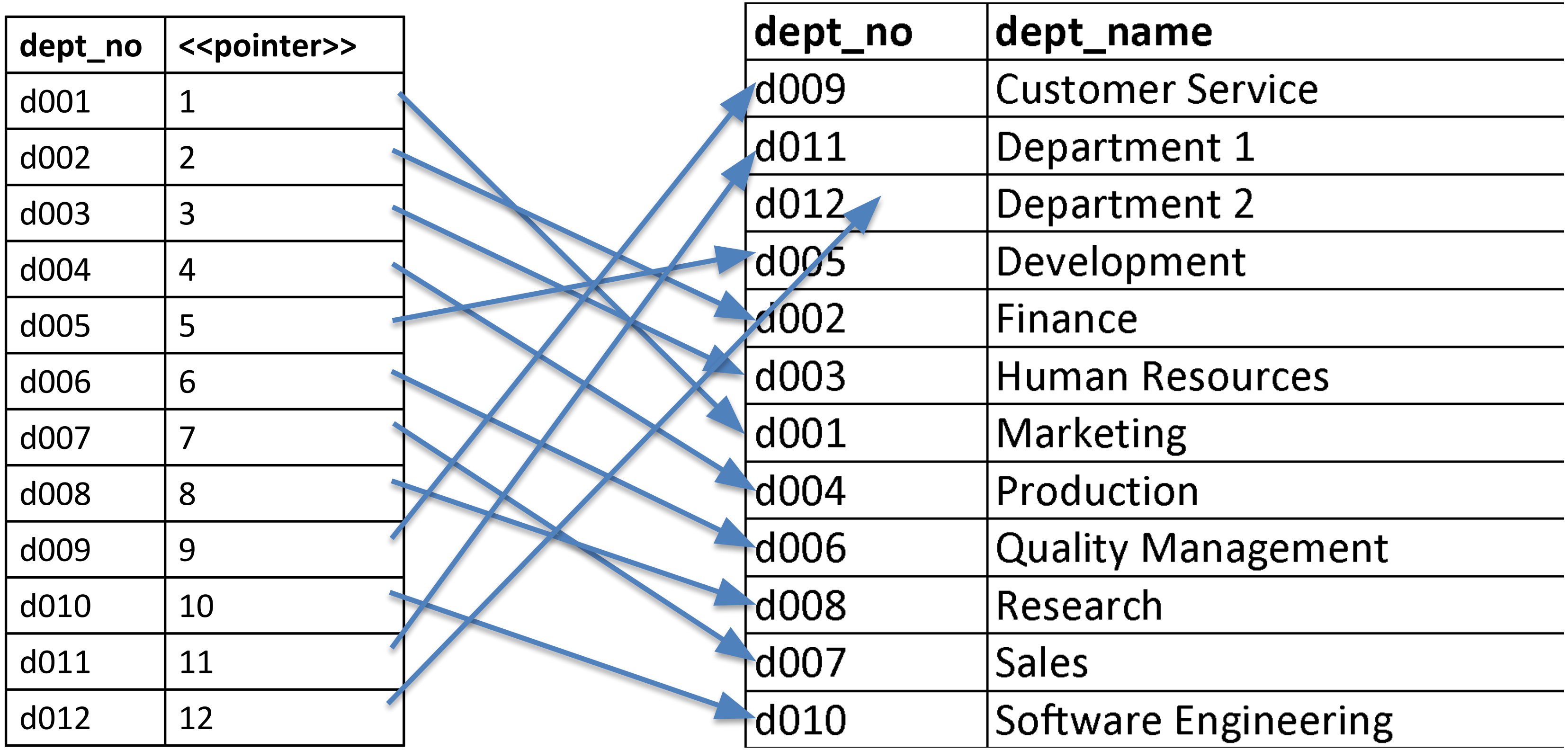
Binary Search

- With a sorted field, a Binary Search has log2 N block accesses.
- Since the data is sorted given a non-key field, the rest of the table doesn’t need to be searched for duplicate values, once a higher value is found.
- The performance increase is substantial.

What Is Indexing?

- Indexing is a way of sorting a number of records on multiple fields.
- Creating an index on a field in a table creates another data structure which holds the field value, and pointer to the record it relates to.
- This index structure is then sorted, allowing Binary Searches to be performed on it.

What Is Indexing?



dept_no	<<pointer>>
d001	1
d002	2
d003	3
d004	4
d005	5
d006	6
d007	7
d008	8
d009	9
d010	10
d011	11
d012	12

dept_no	dept_name
d009	Customer Service
d011	Department 1
d012	Department 2
d005	Development
d002	Finance
d003	Human Resources
d001	Marketing
d004	Production
d006	Quality Management
d008	Research
d007	Sales
d010	Software Engineering

Indexing Downsides

- Indexes require additional space on the disk
- Since the indexes are stored together in a table using the MyISAM engine, this file can quickly reach the size limits of the underlying file system if many fields within the same table are indexed.

- Let’s assume that we have the following MySQL table schema:

Field name	**Data type**	**Size on disk**
id (primary key)	Unsigned INT	4 bytes
firstName	Char(50)	50 bytes
lastName	Char(50)	50 bytes
emailAddress	Char(100)	100 bytes

- This sample table contains five million rows
- It is unindexed.

Example 1

- Sample table of **r** = 5,000,000 records of a fixed size
- Record length of **R** = 204 bytes (4 + 50 + 50 + 100)
- Stored in a table using the MyISAM engine which is using the default block size **B** = 1,024 bytes.
- The blocking factor of the table would be **bfr** = (B/R) = 1024/204 = 5 records per disk block.
- The total number of blocks required to hold the table is **N** = (r/bfr) = 5000000/5 = 1,000,000 blocks.

http://stackoverflow.com/questions/1108/how-does-database-indexing-work

Example 1

- A linear search on the **id** field would require an average of N/2 = **500,000 block accesses** to find a value given that the id field is a key field.
- But since the **id** field is also sorted (primary key) a binary search can be conducted requiring an average of log2 1000000 = 19.93 = **20 block accesses**.

http://stackoverflow.com/questions/1108/how-does-database-indexing-work

Example 1

- Now the firstName field is not sorted, so a binary search is impossible
- The values of firstName are not unique
- The table will require searching to the end for an exact **N = 1,000,000 block accesses**.
- It is this situation that indexing aims to correct.

http://stackoverflow.com/questions/1108/how-does-database-indexing-work

Example 1

- Given that an index record contains only the indexed field and a pointer to the original record, it stands to reason that it will be smaller than the multi-field record that it points to.
- So the index itself requires fewer disk blocks that the original table, which therefore requires fewer block accesses to iterate through.

http://stackoverflow.com/questions/1108/how-does-database-indexing-work

Example 1

- The schema for an index on the firstName field is outlined below:

Field name	Data type	Size on disk
firstName	Char(50)	50 bytes
(record pointer)	special	4 bytes

Example 2

Field name	Data type	Size on disk
firstName	Char(50)	50 bytes
(record pointer)	special	4 bytes

- Sample table of r = 5,000,000 records
- Index record length of R = 54 bytes (50 + 4)
- Default block size B = 1,024 bytes.

http://stackoverflow.com/questions/1108/how-does-database-indexing-work

Example 2

- The blocking factor of the index would be bfr = (B/R) = 1024/54 = **18** records per disk block (**5** records in unindexed table).
- The total number of blocks required to hold the table is N = (r/bfr) = 5000000/18 = **277,778** blocks (**1,000,000** blocks in unindexed table).

http://stackoverflow.com/questions/1108/how-does-database-indexing-work

Example 2

- Now a search using the firstName field can utilize the index to increase performance.
- This allows for a binary search of the index with an average of log2 277778 = 18.08 = **19 block accesses**.
- To find the address of the actual record, which requires a further block access to read, bringing the total to 19 + 1 = **20 block accesses**, vs. the **277,778 block accesses** required by the non-indexed table.

http://stackoverflow.com/questions/1108/how-does-database-indexing-work

MySQL Indexes

- Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees.
- Indexes on spatial data types use R-trees
- MEMORY tables also support hash indexes.

http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

MySQL uses indexes for these operations:

- To find the rows matching a WHERE clause quickly.
- To eliminate rows from consideration. If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows.

http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

MySQL uses indexes for these operations:

- To retrieve rows from other tables when performing joins. MySQL can use indexes on columns more efficiently if they are declared as the same type and size. In this context, VARCHAR and CHAR are considered the same if they are declared as the same size. For example, VARCHAR(10) and CHAR(10) are the same size, but VARCHAR(10) and CHAR(15) are not.

http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

MySQL uses indexes for these operations:

- For comparisons between non-binary string columns, both columns should use the same character set. For example, comparing a utf8 column with a latin1 column precludes use of an index.

http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

MySQL uses indexes for these operations:

- Comparison of dissimilar columns (comparing a string column to a temporal or numeric column, for example) may prevent use of indexes if values cannot be compared directly without conversion.
- Suppose that a numeric column is compared to a string column. For a given value such as 1 in the numeric column, it might compare equal to any number of values in the string column such as '1', ' 1', '00001', or '01.e1'. This rules out use of any indexes for the string column.

http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

MySQL uses indexes for these operations:

- To find the MIN() or MAX() value for a specific indexed column key_col. This is optimized by a preprocessor that checks whether you are using WHERE key_part_N = constant on all key parts that occur before key_col in the index. In this case, MySQL does a single key lookup for each MIN() or MAX() expression and replaces it with a constant. If all expressions are replaced with constants, the query returns at once. For example:

```
SELECT MIN(key_part2),MAX(key_part2)
FROM tbl_name
WHERE key_part1=10;
```

http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

MySQL uses indexes for these operations:

- To sort or group a table if the sorting or grouping is done on a usable key (for example, ORDER BY key_part1, key_part2).
- If all key parts are followed by DESC, the key is read in reverse order.

http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html

MySQL Index Types

- UNIQUE
- PRIMARY KEY
- INDEX
- FULLTEXT

MySQL UNIQUE Index

- All rows of the index must be unique.
- The same row may not have identical non-NULL values for all columns in this index as another row.
- UNIQUE indexes can be used:
  - to speed up queries,
  - to enforce restraints on data

http://stackoverflow.com/questions/707874/differences-between-index-primary-unique-fulltext-in-mysql

MySQL UNIQUE Index

- Your database system may allow a UNIQUE index to be applied to columns which allow NULL values, in which case two rows are allowed to be identical if they both contain a NULL value.
- Depending on your application, however, you may find this undesirable: if you wish to prevent this, you should disallow NULL values in the relevant columns.

http://stackoverflow.com/questions/707874/differences-between-index-primary-unique-fulltext-in-mysql

MySQL PRIMARY KEY Index

- Acts exactly like a UNIQUE index, except:
  - it is always named 'PRIMARY’
  - there may be only one on a table (and there should always be one; though some database systems don't enforce this).

http://stackoverflow.com/questions/707874/differences-between-index-primary-unique-fulltext-in-mysql

MySQL PRIMARY KEY Index

- A PRIMARY index is intended as a primary means to uniquely identify any row in the table
- Unlike UNIQUE it should not be used on any columns which allow NULL values.
- Your PRIMARY index should be on the smallest number of columns that are sufficient to uniquely identify a row.
- Often, this is just one column containing a unique auto-incremented number or auto-generated value (GUID or UUID)

http://stackoverflow.com/questions/707874/differences-between-index-primary-unique-fulltext-in-mysql

MySQL INDEX Index

- INDEX refers to a normal non-unique index.
- Non-distinct values for the index are allowed, so the index may contain rows with identical values in all columns of the index.
- These indexes don't enforce any restraints on your data so they are used only for making sure certain queries can run quickly.

http://stackoverflow.com/questions/707874/differences-between-index-primary-unique-fulltext-in-mysql

MySQL FULLTEXT Index

- FULLTEXT indexes are different from all of the above
- Their behavior differs significantly between database systems.
- FULLTEXT indexes are only useful for full text searches done with :
  - MATCH() / AGAINST() clause in MySQL
  - CONTAINS() clause in MSSQL

Creating Indexes in MySQL

```
CREATE [UNIQUE|FULLTEXT] INDEX index_name
    [index_type]
    ON tbl_name (index_col_name,...)
    [index_type]
index_col_name:
    col_name [(length)] [ASC | DESC]
index_type:
    USING {BTREE | HASH}
```

Creating **INDEX** Indexes in MySQL

```
CREATE INDEX idx_dept_name ON employees.departments (dept_name);
```

CREATE INDEX command

Name of the index. Note that it is convention to begin index names with **idx_**

Name of the table on which you are creating an index

Name of the field (attribute) on which you are creating an index

Creating **UNIQUE** Indexes in MySQL

```
CREATE UNIQUE INDEX idx_dept_id
ON employees.departments (dept_no);
```

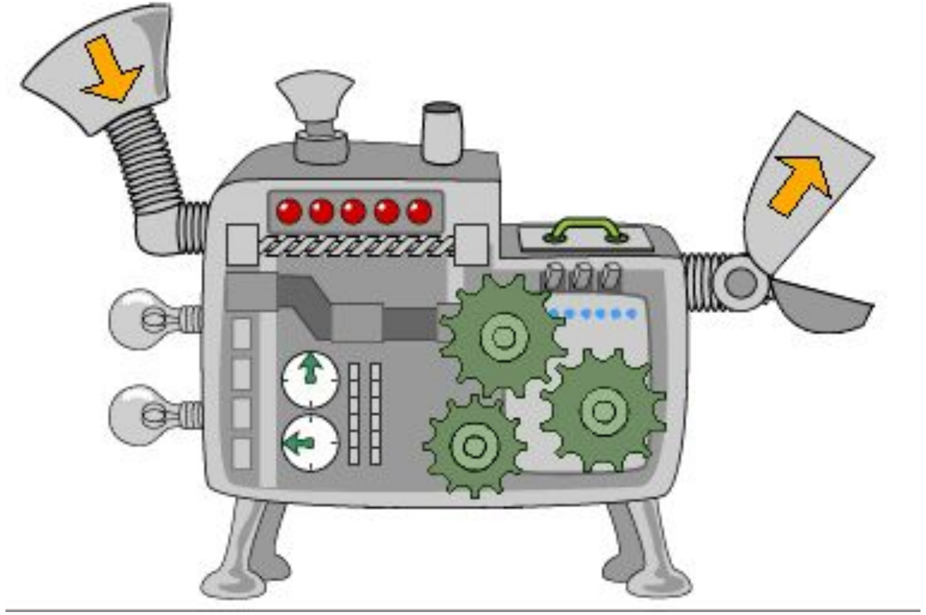
Creating **FULLTEXT** Indexes in MySQL

```
CREATE FULLTEXT INDEX idx_dept_name_fulltext
ON employees.departments (dept_name);
```

Indexes

Functions

Functions



Stored Functions

- **Stored Function** is a stored program that returns a **single value**.
- You can use stored functions to **encapsulate** common formulas or business rules that may be **reusable** among SQL statements or stored programs.
- You can use a stored function in SQL statements wherever an expression is used. This helps improve the readability and maintainability of the procedural code.

Stored Functions

- Functions cannot perform permanent environmental changes to SQL Server (i.e. no INSERT or UPDATE statements allowed).
- A Function can be used inline in SQL Statements if it returns a scalar value or can be joined upon if it returns a result set

Stored Functions

Function

SELECT * FROM orders WHERE **YEAR**(shippedDate) = 2003;

Create Function Syntax

```
CREATE FUNCTION function_name(param1,param2,…)
  RETURNS datatype
  [NOT] DETERMINISTIC
    statements
```

Create Function Syntax

- Specify the name of the stored function after CREATE FUNCTION keywords.

```
CREATE FUNCTION addTwoNumbers
```

- List all parameters of the function.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)
```

Create Function Syntax

- Specify the data type of the return value in the RETURNS statement. It can be any valid MySQL data type.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)
            RETURNS double
```

Create Function Syntax

- For the same input parameters, if the function returns the same result, it is considered deterministic and not deterministic otherwise.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)
          RETURNS double
          DETERMINISTIC
```

Create Function Syntax

- For the same input parameters, if the function returns the same result, it is considered deterministic and not deterministic otherwise.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)
            RETURNS double
            DETERMINISTIC
```

- You have to decide whether a stored function is deterministic or not. If you declare it incorrectly, the stored function may produced an unexpected result, or the available optimization is not used which degrade the performance.

Create Function Syntax

- Fifth, you write the code in the statements section. It can be a single statement or a compound statement. Inside the statements section, you have to specify at least one **RETURN** statement.
- The RETURN statement returns a value to the caller. Whenever the RETURN statement is reached, the stored function’s execution is terminated immediately.

Create Function Syntax

```
DELIMITER $$

CREATE FUNCTION addTwoNumbers(num1 double, num2 double)
	RETURNS double
	DETERMINISTIC
BEGIN
	DECLARE sumOfTwoNumbers double;

	sumOfTwoNumbers = num1 * num2;

	RETURN (sumOfTwoNumbers);
END
```

Create Function Syntax

```
DELIMITER $$

CREATE FUNCTION customerLevel(p_creditLimit double) RETURNS VARCHAR(10)
        DETERMINISTIC
BEGIN
        DECLARE lvl varchar(10);

        IF p_creditLimit > 50000 THEN
                SET lvl = 'PLATINUM’;
        ELSEIF (p_creditLimit <= 50000 AND p_creditLimit >= 10000) THEN
                SET lvl = 'GOLD';
        ELSEIF p_creditLimit < 10000 THEN
                SET lvl = 'SILVER';
        END IF;

        RETURN (lvl);
END
```