DATABASE SYSTEMS PROJECT
Search Engine Database

PIERPAOLO MASELLA - 710076

A.Y. 2018/2019

Project developed in A.Y. 2019/2020

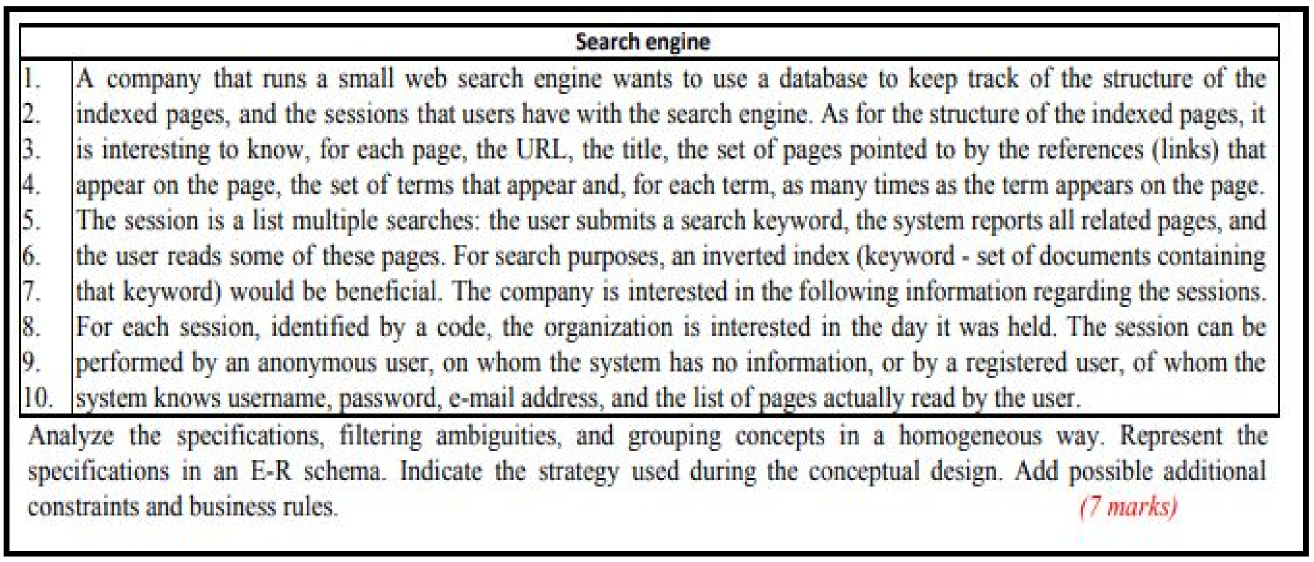
Contents

Introduction

The project presented in this paper has as main goal the implementation from scratch of an *object relational database* using *Oracle 11g;* the **secondary goal** has been represented by the implementation of a web site which uses the database created as a backend.
The steps will explained following this schema:

- Conceptual design
- Logical design
- Logical implementation
- Physical design
- Website implementation

Conceptual design

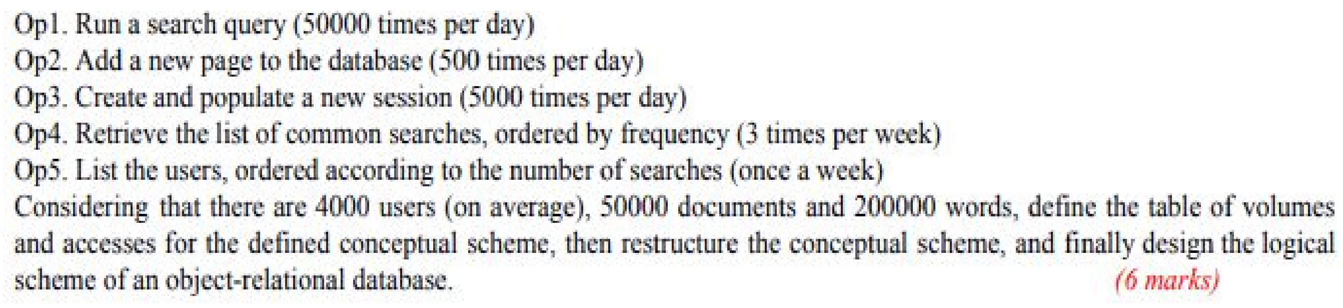


Search engine

A company that runs a small web search engine wants to use a database to keep track of the structure of the indexed pages, and the sessions that users have with the search engine. As for the structure of the indexed pages, it is interesting to know, for each page, the URL, the title, the set of pages pointed to by the references (links) that appear on the page, the set of terms that appear and, for each term, as many times as the term appears on the page. The session is a list multiple searches: the user submits a search keyword, the system reports all related pages, and the user reads some of these pages. For search purposes, an inverted index (keyword - set of documents containing that keyword) would be beneficial. The company is interested in the following information regarding the sessions. For each session, identified by a code, the organization is interested in the day it was held. The session can be performed by an anonymous user, on whom the system has no information, or by a registered user, of whom the system knows username, password, e-mail address, and the list of pages actually read by the user.

Analyze the specifications, filtering ambiguities, and grouping concepts in a homogeneous way. Represent the specifications in an E-R schema. Indicate the strategy used during the conceptual design. Add possible additional constraints and business rules. *(7 marks)*

And the basic operations needed by the requirements are:



Op1. Run a search query (50000 times per day)
Op2. Add a new page to the database (500 times per day)
Op3. Create and populate a new session (5000 times per day)
Op4. Retrieve the list of common searches, ordered by frequency (3 times per week)
Op5. List the users, ordered according to the number of searches (once a week)
Considering that there are 4000 users (on average), 50000 documents and 200000 words, define the table of volumes and accesses for the defined conceptual scheme, then restructure the conceptual scheme, and finally design the logical scheme of an object-relational database. *(6 marks)*

Analysis of the requirements:

Choosing the abstraction level:

The search engine retrieval mechanism will not be represented; linked to this concept we also underline that a *keyword* should not correspond directly to a *term* (a user can search a keyword not present in the db but that will allow them to retrieve a page thanks to similarity measures), but for simplicity they will be treat as the same in the implementation steps described after the logical design.

Synonyms and homonyms

- page and indexed pages are synonyms
- system is a synonym for *search engine*

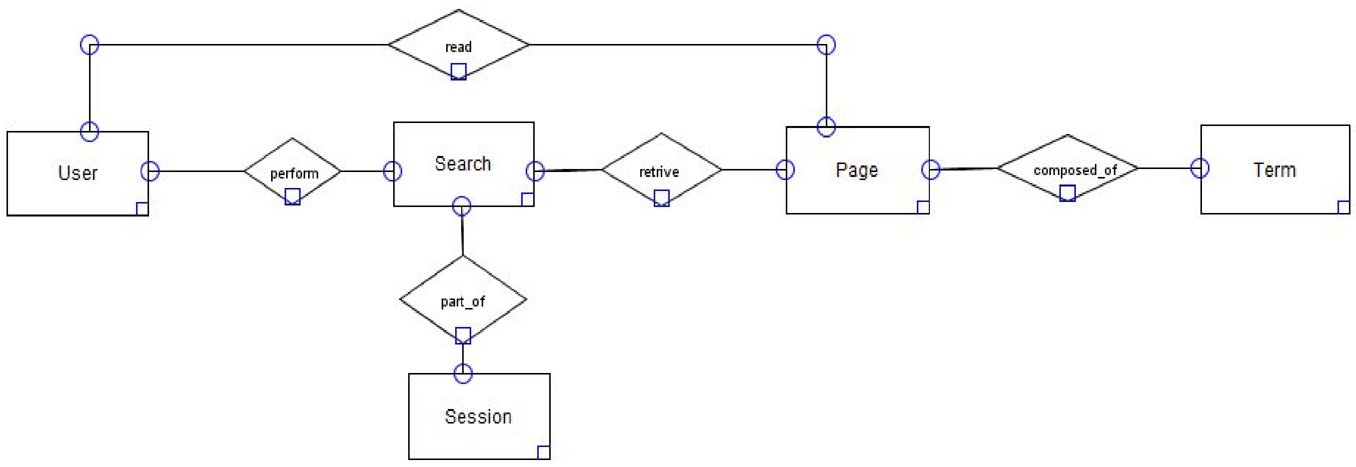
Standardizing phrases and grouping by keyword

- **General phrases:** A company that runs a small web search engine wants to use a database to keep track of the structure of the indexed pages and the sessions that users have with the search engine.
  The company is interested in following information regarding the session.
- **Session**: The session is a list of multiple searches. The session can be performed by *anonymous users* or *registered users*.
  For the session we are interested in representing it's code (identifier) and the data in which it started.
- **User:** The user submits a search keyword, the system reports all *related pages*, the user reads one or more pages.
  The user can be a registered user or anonymous user. For anonymous users we have no information about. For registered users we are interested in representing username, password, e-mail address, and the list of pages actually read.
- **Pages**: For the pages we are interested in representing the URL (identifier), the title, the set of terms that appear in the page, the links to other pages.
- **Term:** For the term we are interested in representing its name.
  An inverted index approach is used to store the terms.

Design

The first thing that must be done is that of the conceptual design. To do this, a **hybrid approach** will be used. In this sense, first we have to create a skeleton of the conceptual schema.

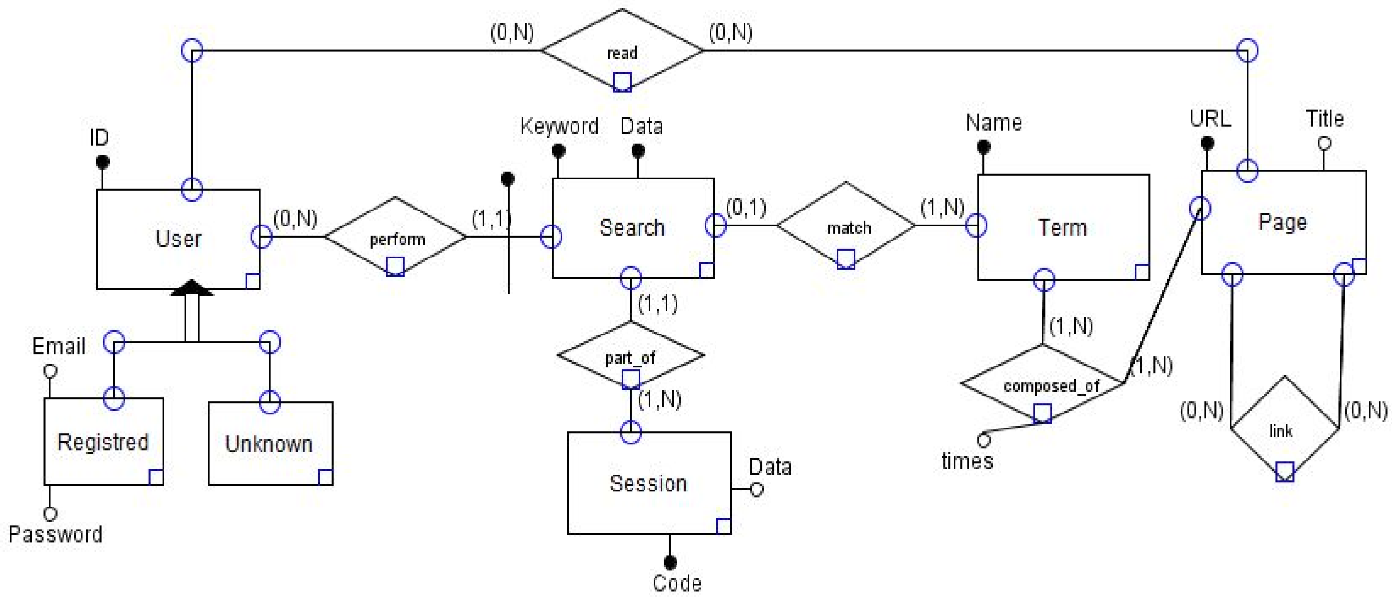
Skeleton schema



The proposed schema is a *skeleton schema* which represents the main entities found in the requirements analysis.

First refinement and final schema

In this phase, we are refining the schema proposed previously, in order to model more in detail the entities and the relationships between them.



It is possible to note that the schema represents a generalization for users, in order to model the two kinds of users that are required: *registered user* and *anonymous user*; following the requirements, attributes and identifiers have been introduced.
A recursive relationship *"link"* has been introduced to model the links between pages, so that a page can optionally contain links to other pages.
It is possible to note that the *inverted index* approach has been modelled introducing an attribute *times* on *composed_of* relationship.
Another important aspect is that relationship *"match"* has been introduced instead of "retrieve", in order to model the fact that the keyword of the search matches at most one term **(assumption: for simplicity the system accept one keyword in input that will match with one term, or nothing).**

Derivation rules

- the number of pages read by user can be obtained counting the occurrences of "read" relationship regarding that user
- the number of searches performed by user can be obtained counting the occurrences of "perform" relationship regarding that user

Constraint rules

It is possible to deepen the constraints that cannot be expressed in the conceptual schema, but that will be modelled in the logical design phase and implementation phase:

- A user can read the page only if it has been retrieved by the search engine; this relationship can also be interpreted as historical "reads" storage.
- A session will contain more than one search made by the same user while the user is connected to the system
- The same unknown user can have different ids associated, because of course we do not know information about them.

Logical design

Observations: in the conceptual scheme it is not possible to state that a session, if it does not exist, should be created after a search; if, on the other hand, a search is done in the same time frame (e.g. while logged in to the site) then the search is added to a pre-existing search session.

Volumes Table

Concept	Type	Volume	Explanation
User	E	4000	Given from the specification
Registered	E	2000	Half of the users is assumed to be registered
Unknown	E	2000	Half of the users is assumed to be unknown
Perform	R	40000	Consistent with search
Search	E	40000	A user performs on average 10 research
Match	R	36000	Assuming that the system matches 9 terms out of 10
Read	R	80000	Assuming that a user reads, on average, the first 2 results (pages) returned, we have 4000 users * 10 searches * 2 pages read
part_of	R	20000	Consistent with session
Session	E	20000	A session is composed of 2 research on average
Term	E	200000	Given from the specification
Composed_of	R	2500000	assuming that, on average, each page contains 50 unique terms, and so, each term appears on average in 4 pages
Page	E	50000	Given from the specification
Link	R	100000	Assuming that one page has 2 link on average

Access Table and redundancy analysis

- *Operation 1:* Run a search query (50000 times per day) - Interactive
  - assuming that the user is already in a session (so it's not the first search, because that scenario will be discussed thanks to *operation 3*), and so we know the session id and the user id
- *Operation 2*: Add a new page to the database (500 times per day) - Interactive
- *Operation 3*: Create and populate a new session (5000 times per day) - Interactive
  - it means that a user run a query in a new session, performing the first query of that session
- *Operation 4*:Retrieve the list of common searches, ordered by frequency (3 times per week) - Batch
- *Operation 5:* List the users, according to the number of searches(once a week) - Batch

  OTHER OPERATIONS (not request)
- *Operation 6:* Registration of a new user (10 times per day) - Interactive

- *Operation 7:* Update of the dictionary of terms inserting a new term due to the creation of page with a new and unknown term - Batch **Contained in the operation 2**

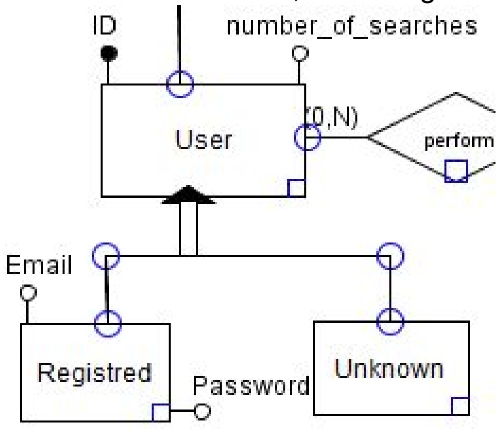
OP.	**CONCEPTS**	**CONCEPT TYPE**	**ACCESSES**	**ACCESS TYPE**	**EXPLANATION**
1					
	User	E	1	R	Read the type of user
	Perform	R	1	W	Write the user id and the new search
	Search	E	1	W	write data about the new search
	part_of	R	1	W	connect the search to existing session of which the id is already known
	term	E	1	R	We search the term for matching it with the keyword
	match	R	1	W	Write the search that matched a term with the keyword
	composed_of	R	4	R	On average, we stated that a term appears in 4 pages
	page	E	4	R	We read the URL of the pages
	read	R	2	R	On average, a user reads 2 of the pages returned
2					
	Page	E	1	W	Insert details of the new page
	Link	R	2	W	On Average, 2 link per page
	Term	E	49	R	Search for the terms that are already present in the system, but assuming that everytime a new page contains on average a new term
	Term	E	1	W	Write the new term which did not match with the terms in the database
	Composed_of	R	50	W	Write the total number of term that are present in the page
3					
	User	E	1	R	read user id
	Perform	R	1	W	
	Search	E	1	W	write search details
	Session	E	1	W	create new session

	part_of	R	1	W	link session and search
	Term	E	1	R	search for the term,at most one result
	match	R	1	W	write the matching between keyword searched and term present in the database
	composed_of	R	4	R	on average, each term in contained in 4 pages
	Page	E	4	R	read the urls of the page retrieved
	Read	R	2	W	Consistent with the assumption that a user reads 2 pages on average
4					
	Search	E	40000	R	Read all the searches and group by keywords to count the occurences
5					
	User	E	4000	R	Read the user ID
	Perform	R	40000	R	count the occurences of the searches made by each user
6					
	User	E	1	W	Creation of the user

Redundancies analysis

Op. 5 could benefit in terms of costs introducing the attribute "number_of_searches"; in the following lines both the implementations' costs are discussed and compared.

Operation 5: List the users, according to the number of searches(once a week) - Batch



ACCESSES TABLE WITH REDUNDANCY

OP.	**CONCEPTS**	**CONCEPT TYPE**	**ACCESSES**	**ACCESS TYPE**	**EXPLANATION**

5					
	User	E	4000	R	Read the user ID

- **With Redundancy**
  4000 *1 = 4000 accesses in read
  4000*4 Byte = 16KB
- **Without Redundancy**
  44000*1 = 44000 accesses in read

So, it’s useful to maintain the redundancy, considering the cost.

Generalizations

No change has been done in this phase

Partitioning and merging

We can observe that “search” and “session” are involved together in the same operations: it could be useful to merge them into the entity search.

The new access table, for the *Operation 1*, that is “Run a search query “and for the *Operation 3*, that is “Create and populate a new session “, will be reported below.
With this merge, we will have 1 write less for *Operation 1* and 2 writes less for *Operation 3*

OP.	**CONCEPTS**	**CONCEPT TYPE**	**ACCESSES**	**ACCESS TYPE**	**EXPLANATION**
1					
	User	E	1	R	Read the type of user
	Perform	R	1	W	Write the user id and the new search
	Search	E	1	W	write data about the new search
	term	E	1	R	We search the term for matching it with the keyword
	match	R	1	W	Write the search that matched a term with the keyword
	composed_ of	R	4	R	On average, we stated that a term appears in 4 pages
	page	E	4	R	We read the URL of the pages
	read	R	2	R	On average, a user reads 2 of the pages returned
3					

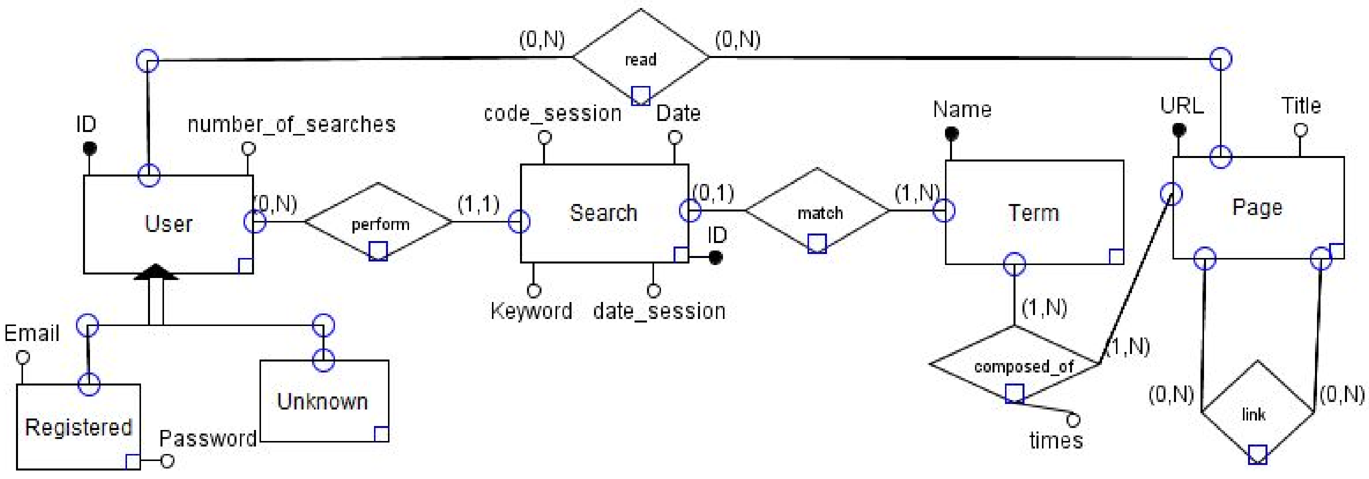
	User	E	1	R	read user id
	Perform	R	1	W	
	Search	E	1	W	write search details
	Term	E	1	R	search for the term,at most one result
	match	R	1	W	write the matching between keyword searched and term present in the database
	composed_ of	R	4	R	on average, each term in contained in 4 pages
	Page	E	4	R	read the urls of the page retrieved
	Read	R	2	W	Consistent with the assumption that a user reads 2 pages on average

Choice of main identifier

On search it has been decided to define a new identifier ID removing the foreign key composed with the two primary key which were date and keyword.

Final schema

So, the final schema will be:



Object-Relational implementation

The source code of the implementation of the designed database can be found in the folder attached, called *scripts_cinema_app,* organized as follows:

- *scripts_cinema_app* (attached folder)
    - types declaration.sql
    - tables declaration.sql
    - triggers.sql
    - stored_functions.sql
    - *procedures* (folder)
        - populating links (page_tab).sql
        - populating searches.sql
        - stored procedures.sql

in the following table will be reported only the implementations of types and tables, in order to allow the reader to observe the mapping between the final logical schema and the implementation.

Types Implementation	**Table Implementation**
/ create or replace type page_t as object (url VARCHAR2(2100), title VARCHAR2(150), links pages_nt, is_indexed number); / create or replace type pages_nt as TABLE of ref page_t; / create or replace type history_nt_type as TABLE of ref page_t; / create or replace type user_t as object (user_id NUMBER, searches_n NUMBER, pages_read history_nt_type) NOT FINAL NOT INSTANTIABLE; / create or replace type registered_t UNDER user_t (Email VARCHAR2(50), psw VARCHAR2(50), name_user VARCHAR2(30), Surname VARCHAR2(30), constructor function registered_t(self in out NOCOPY registered_t, email VARCHAR2, psw	create table page_tab of page_t (url PRIMARY KEY, title NOT NULL, is_indexed default 0) nested table links store as pages_nt_table; / create table user_tab of user_t (user_id PRIMARY KEY, searches_n DEFAULT 0) nested table pages_read store as user_history_nt; / create table search_tab of search_t (search_id PRIMARY KEY, code_session NOT NULL, search_time NOT NULL, keyword NOT NULL, user_associated NOT NULL); / create table term_tab of term_t (term_name PRIMARY KEY) nested table inverted_index store as inverted_index_nt_table; / end; /

```
VARCHAR2, name_user VARCHAR2, surname
VARCHAR2)
return self as result);

/
--imlplementazione costruttore per evitare duplicati,
poichè col trigger è impossibile
create or replace
TYPE BODY registered_t AS
   CONSTRUCTOR FUNCTION registered_t(self in
out NOCOPY registered_t, email VARCHAR2, psw
VARCHAR2, name_user VARCHAR2, surname
VARCHAR2)
   return self as result is
   conflicts number;
   begin
       select count(*) into conflicts from user_tab u
where value(u) is of (registered_t) and
treat(value(u) as registered_t).email=email;
     if conflicts>0 then
              raise_application_error('-20010', 'Email
already exists');
     else
       self.user_id:=null;
       self.searches_n:=0;
       self.pages_read:=null;
       self.email:=email;
       self.psw:=psw;
       self.name_user:=name_user;
       self.surname:=surname;
     end if;
     return;
   end;
end;

/

create or replace type unknown_t UNDER user_t(
constructor function unknown_t(self in out
NOCOPY unknown_t)
return self as result);
/
create or replace
TYPE BODY unknown_t AS
    CONSTRUCTOR FUNCTION unknown_t(self in
out NOCOPY unknown_t)
   return self as result is
   begin
       self.user_id:=null;
       self.searches_n:=0;
       self.pages_read:=null;
     return;
   end;
end;
```

```
/
create or replace type inverted_index_t as object
(page ref page_t,
occurrences number);
/
create or replace type inverted_index_nt is table of
inverted_index_t;

/
create or replace type term_t as object
(term_name varchar(20),
inverted_index inverted_index_nt);

/
create or replace type search_t as object
(search_id NUMBER,
code_session NUMBER,
search_time TIMESTAMP,
keyword VARCHAR2(20),
user_associated ref user_t,
term ref term_t);
/
--funzionale alla procedura "searchword" per
istanziare una table
create or replace TYPE word_found as object
(term varchar2(30),
occurrences number,
MEMBER procedure increment_occurrences); --per
creare la  table
/
--implementazione metodo di word_found
create or replace type body word_found as
member procedure increment_occurrences is
begin
occurrences:=occurrences+1;
end;
end;
/
create or replace TYPE words_in_page_type is
table of word_found;
```

Managing constraints

In order to maintain the constraints underlined in the requirements and explained during the design phases, it has been necessary to implement several triggers (implementation can be found in the file *triggers.sql*):

- auto-increment the ids (primary keys) on tables *search_tab* and *user_tab*
- creation of a new session id on table *search_tab*
- implementation of the **inverted index** on the table page_tab using an auxiliary function implemented from scratch called *search_word*

To guarantee the uniqueness of the email, which is characteristic of registered user, it has been necessary to override the constructor (as you can notice reading the implementation reported in the previous table).
Some of the implementation reported required to be **autonomous transactions** in order to overcame the problem of **mutating tables** during complex operations which involve several tables.

Population of database

The population of the database can be summarized as:

- user_tab (4000 instances = 2000 registered users + 2000 unknown users)
    - Performed with python, using a dataset created from scratch merging two datasets of surnames and names of Italian people from Lecce (source code can be found in the folder "*populating database pycharm"*
    - Show the inserted values with the following query
      select treat(value(u) as registered_t).name_user as name,treat(value(u) as registered_t).surname as surname,
      treat(value(u) as registered_t).email as email, treat(value(u) as registered_t).psw as psw
      from user_tab u where value(u) is of (registered_t);

- page_tab (50000 instances)
    - Performed with python, using a dataset downloaded from Kaggle containing *medium* website posts; the dataset used after the processing of the original one, contains a urls and titles for the pages(source code can be found in the folder "*populating database pycharm"*; **term table is populated thanks to the trigger that manage the inverted index**
    - Show the inserted values with the following query
      select deref(column_value).url from table(select links from page_tab');
- Popolamento search_tab
    - Performed using a procedure implemented in pl/sql, querying the data inserted both into *user_tab* and *page_tab*
    - Show the inserted values with the following query
      select s.search_id, s.keyword, deref(s.user_associated).user_id as user_id, deref(s.term).term_name as math_term from search_tab s;

Physical design

at this stage the performance of the system is estimated, trying to optimize them through physical interventions
The performances, in this physical part, will be evaluated using the "set timing on" command which allows us to check how long an operation is performed. By comparing time, for example, whether or not indexes are present, we could see whether or not these might be useful depending on the operation.

Operations 1 and 3

"Operation 1: Run a search query" and "*Operation: 3* Create and populate a new session" are strictly linked each other; the analysis have been the following

- ○ assuming that the user is already in a session (so it's not the first search, because that scenario will be discussed thanks to *operation 3*), and so we know the session id and the user id

- ○ In this operation, the time measure is closely related to the type of keyword sought. For example, trying to find the keyword "*and"* in 15989 documents requires 4,859 ms without an index.
- ○ Instead, by querying for all documents that contain the *word "and"* with a string matching approach (and so skipping insertion into table search_tab and the increment of user searches) performances cost about 4 times more at a cost of over 20 sec, demonstrating that the inverted index approach is efficient.

These operations can't be optimized more.

Operation 2

Operation 2: Add a new page to the database (500 times per day)

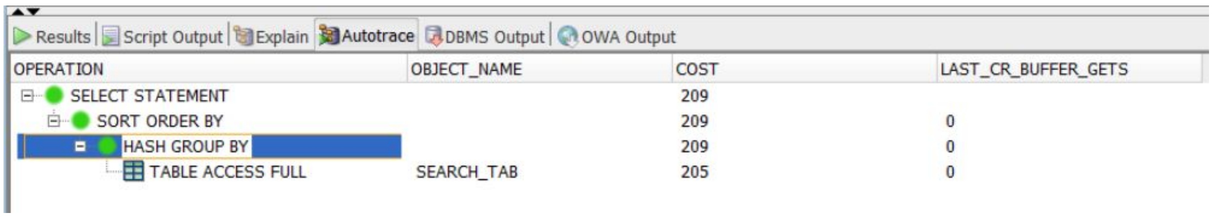
This operation is a simple insert but involves the inverted index trigger; thanks to the usage of refs and thanks to the fact that the term is the primary key in term_tab, there is no need of improvement.

Operation 4

Operation 4:Retrieve the list of common searches, ordered by frequency (3 times per week) - Batch

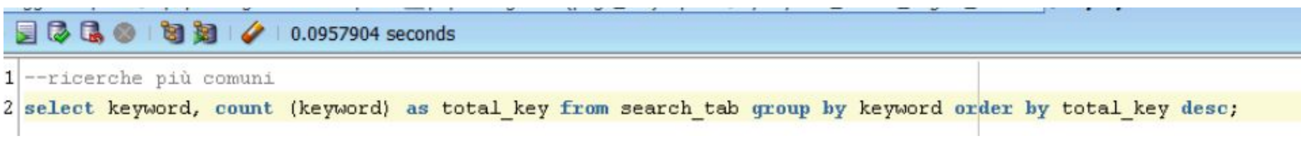
Given the following query
select * from (select keyword, count (keyword) as total_key from search_tab group by keyword order by total_key desc) where rownum <= 50;

To do this we need to count within search_tab all occurrences of a keyword and finally show the most common. In addition, we use rownum to display only the top 50 results decreasingly using the desc.



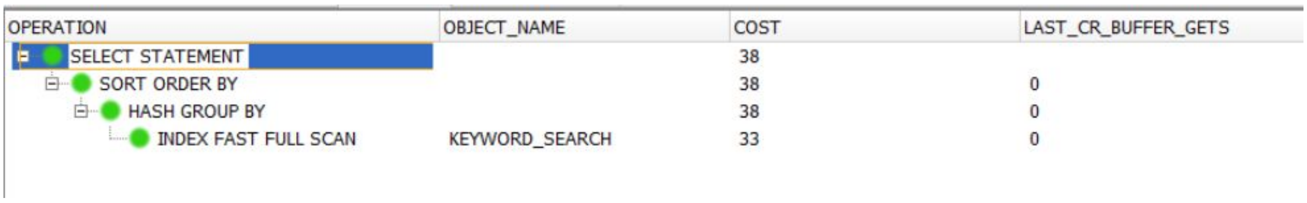
OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		209	
SORT ORDER BY		209	0
HASH GROUP BY		209	0
TABLE ACCESS FULL	SEARCH_TAB	205	0

Looking at the access table, obtained by performing the autotrace on SQL Developer, we observe that the **cost of operation 4 is 209** and in terms of time it took **31ms**


0.0957904 seconds

```
--ricerche più comuni
select keyword, count (keyword) as total_key from search_tab group by keyword order by total_key desc;
```

After creating an index on keyword attribute:



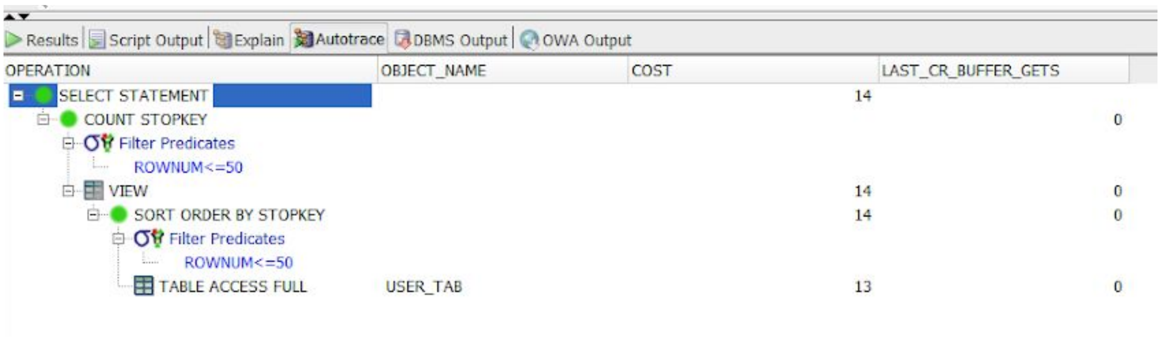
OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		38	
SORT ORDER BY		38	0
HASH GROUP BY		38	0
INDEX FAST FULL SCAN	KEYWORD_SEARCH	33	0

in terms of time still stands at 31 ms but the performance in terms of access suggests keeping the index

Operation 5

- *Operation 5:* List the users, according to the number of searches(once a week) - Batch
  Given the following query:
  select * from(select id_user, searches_n as total_search from user_tab order by total_search desc) where rownum <=50;

  To perform this operation, we need to look within the user_tab for the n_searches attribute and sort that value descendingly. For better display, the first 50 values were also chosen in this case


Results
Script Output
Explain
Autotrace
DBMS Output
OWA Output

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		14	
COUNT STOPKEY			0
Filter Predicates			
ROWNUM<=50			
VIEW		14	0
SORT ORDER BY STOPKEY		14	0
Filter Predicates			
ROWNUM<=50			
TABLE ACCESS FULL	USER_TAB	13	0

Looking at the access table, obtained by autotrace on SQL Developer, we see that the cost of operation 5 is 14 in **32 ms**

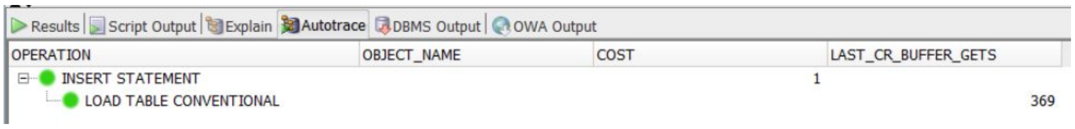
inserting an index in descending order on the searches_n attribute we have the same accesses but half of the time (15ms)
create index searches_of_user_index on user_tab(searches_n desc);

Operation 6a-6b

Operations

- *Operation 6a:* Registration of a new registered user (10 times per day) - Interactive -

    - 31 ms with cost 1 without index



OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
INSERT STATEMENT		1	
LOAD TABLE CONVENTIONAL			369

    - 16 ms with index **defined in *Operation 8***
- Operation 6b : Registration of a new unknown user (10 times per day) - Interactive -
    - In term of time the cost remained 16 ms

Operation 8

Operation 8: Login user
create unique index email_password on user_tab u (treat(value(u) as registered_t).email, treat(value(u) as registered_t).psw);

- 31 ms without index

- 16ms with index

Website

For the execution of the operations listed in the specifications, a GUI has been implemented. In particular the technologies that have been used are: JSP templates and servlets and classical technologies for web development like HTML and CSS.
We have also tried to follow the MVC pattern for the application implemented. So different components have been realized.

For the implementation of the controller component a servlet has been realized. This servlet will manage all the requests done by the user. In particular, it will call different methods implemented in the model component that allows the execution of classical operations like the database insert or login/logout operations.
In other words, this servlet has to receive the requests from the client and call the correct methods implemented in the model component.

The view component is used to present data. For the implementation of this component some JSP pages has been used. This component contains only the code relative to the presentation of data. All the workload is moved to the control and model component.