International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Query Optimization Techniques in Retrieval-Augmented Generation (RAG) Systems

Ayush Aryan[1], Sudhakar Ranjan[2]

[1,2]Student, Department of Computer Science and Engineering, Bennett University, School of Engineering and Technology, Apeejay Stya University, India

Abstract:
Retrieval-Augmented Generation (RAG) helps language models give more accurate answers by using information from external documents. Instead of depending only on what the model learned during training, it can search for relevant information when needed. The quality of the retrieved information is important for getting good results. This paper reviews different ways to improve retrieval, such as rewriting queries, adding related terms, breaking complex questions into smaller parts, generating a sample answer before searching, and refining queries through multiple steps. A simple example is included to show how these methods can improve search results. The paper also compares the techniques and discusses their advantages and disadvantages. Finally, it highlights some common challenges, including slower response times, difficulties in measuring performance, and the possibility of changing the user's original meaning.

Keywords: Retrieval-Augmented Generation, Query Optimization, Query Rewriting, Query Expansion, Large Language Models, Information Retrieval

1. Introduction

Large language models are good at sounding confident, even when they are wrong. That is the basic problem RAG tries to solve. Instead of trusting the model's frozen, training-time knowledge for everything, RAG hands it a small set of retrieved passages now it needs to answer, so the response is at least grounded in something checkable. The idea was formalized by [1,5], and since then it has become close to a default setting up for any LLM application that needs to stay current or speak to a specific domain.

Broadly, it covers any technique that rewrites [8-13], expands, splits up, or otherwise reshapes a query before retrieval, with the single goal of getting more relevant material into the model's context window. In this paper main approaches discusses in use today, tries to put them side by side fairly, and includes a small illustrative example to make the discussion less abstract. Section 2 covers background of the RAG pipeline, Section 3 lays out the taxonomy of techniques, Section 4 works through an example and a comparison, Section 5 talks about where things still go wrong and closes with some thoughts on where this is headed.

2. Background: Where the Query Fits in the RAG Pipeline:

A standard RAG system has three main parts: indexing, retrieval, and generation. Indexing happens first, and it happens offline. The source documents are broken into smaller chunks. Each chunk is converted

into a vector and stored in a vector index. Retrieval happens when a query comes in. The query is also converted into a vector. The system then finds the chunks whose vectors are closest to the query vector. In hybrid setups, a lexical method like BM25 also runs alongside this vector search to catch exact keyword matches. Generation happens last. The retrieved chunks are combined with the original query and passed to the LLM. The LLM then uses this combined input to generate the final answer.

TF-IDF is one of the oldest and most common methods used to retrieve relevant text. It is often used as a baseline when testing newer retrieval techniques in RAG systems. The idea is simple. A term that appears often in a document is likely important to that document. This is called Term Frequency. But if a term appears in almost every document, it loses its value. This is handled using Inverse Document Frequency, which reduces the weight of very common words. In RAG pipelines, TF-IDF can be used to quickly narrow down a large set of documents before sending them to the generation model.

$$\text{TF-IDF}(\mathbf{t}, \mathbf{d}) = \text{TF}(\mathbf{t}, \mathbf{d}) \times \mathbf{log}\left(\frac{\mathbf{N}}{\mathbf{n_t}}\right) \qquad (1)$$

Here, $\mathbf{f_{t,d}}$is how many times term **t**appears in document **d**. **N**is the total number of documents. $\mathbf{n_t}$is the number of documents that contain term **t**. The query is really the only lever a user has over what gets retrieved. If it's too short or ambiguous, recall suffers; if it's overloaded with multiple questions at once, the retriever tends to satisfy none of them particularly well. Gao et al. (2023) make a similar point in their survey [3], noting that a single literal query often isn't enough to gather the context a complex question needs, and that generation models which lean too heavily on weak retrieved context end up producing answers that just restate whatever was retrieved rather than reasoning over it properly. Table 1 shows the types of RAG and best for.

Table 1: RAG types and best for

RAG Type	Best For
Naive RAG	Simple Q&A, prototypes
Advanced RAG	Better accuracy, production-ready
Modular RAG	Flexible, customizable pipelines
Graph RAG	Relationship-heavy data
Hybrid RAG	Cases where exact match and semantic search are both needed
Agentic RAG	Multi-step, complex reasoning
Self-RAG	Self-correcting, high-accuracy needs
Corrective RAG	Noisy or unreliable data sources

Most taxonomies split RAG into a "naive" version, where the raw query goes straight to the retriever in one pass, and an "advanced" or "modular" version, which adds optimization steps before and after retrieval [1-4]. Query optimization techniques sit mostly on the pre-retrieval side of that split, though a few methods discussed below, especially the iterative ones [6-7], blur the line by looping back and forth between retrieval and generation more than once.

3. Architecture Overview

Before going through each technique individually, it's worth stepping back and looking at where query optimization sits inside a full RAG pipeline. Figure 1 lays this out end to end: a raw query first passes through a query analyzer or router, which decides whether optimization is needed at all and, if so, which

technique (or combination of techniques) to apply. The optimized query is then sent to a retriever that searches a vector index, often alongside a lexical index in hybrid setups, and whatever comes back is passed through a re-ranker before finally reaching the LLM generator.

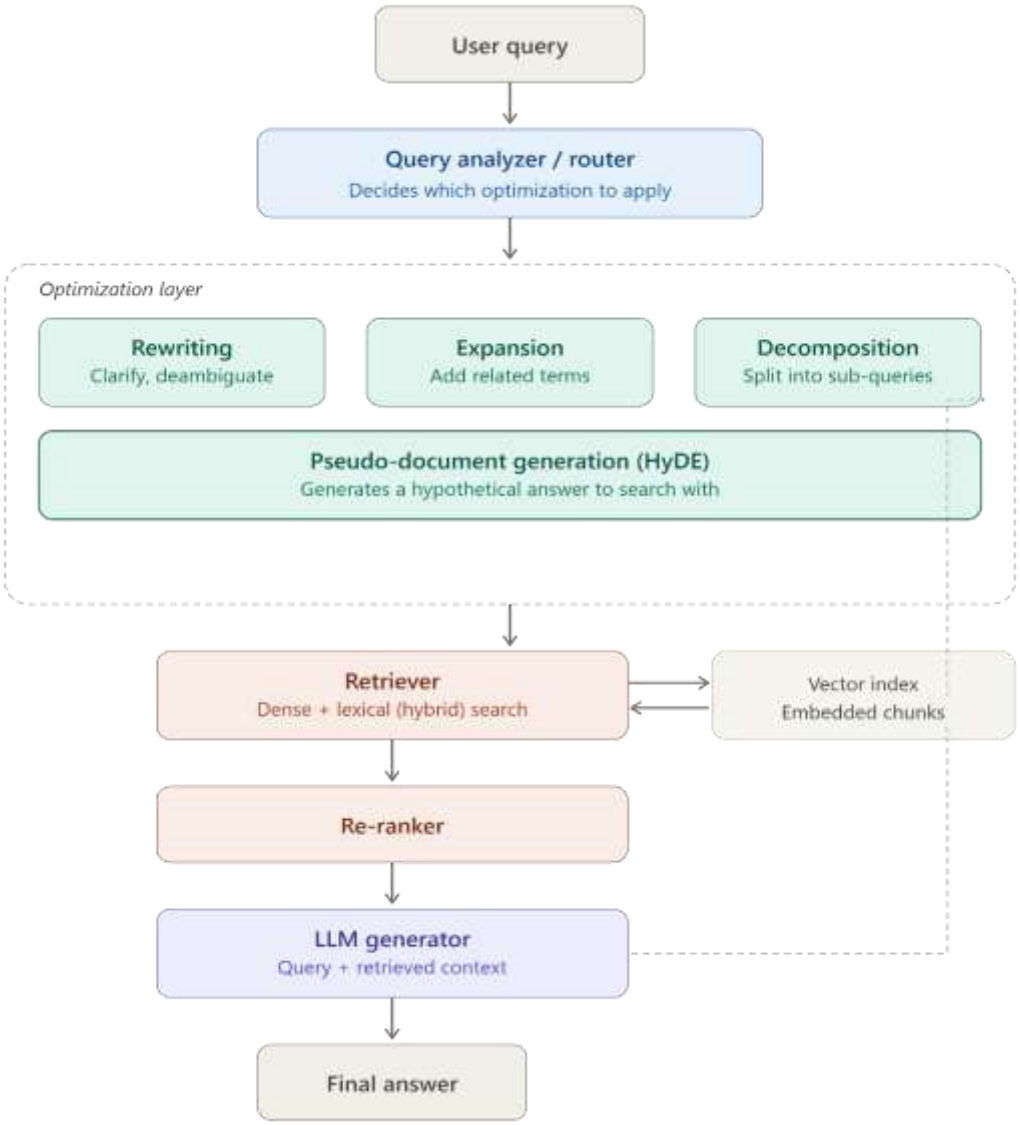


Figure 1. End-to-end RAG architecture.

Figure 1. showing where query optimization techniques sit relative to retrieval and generation. The dashed loop on the right represents iterative, feedback-driven refinement, where the generator (or an evaluator watching it) can send the query back through the optimization layer if the retrieved evidence turns out to be insufficient. A couple of things about this layout are worth calling out. First, the four boxes inside the dashed "optimization layer" are not meant to run in sequence on every query, that would be wasteful. The router at the top is what decides which path (or paths) a given query needs; a short, well-formed query might skip the optimization layer entirely and go straight to the retriever, while a vague multi-hop question might pass through decomposition and then HyDE before retrieval even starts. Second, the feedback loop on the right is optional and only kicks in for the iterative/feedback-driven techniques discussed in Section 4.5, most RAG deployments today still run a single optimization pass and skip this loop for latency reasons.

4. Query Optimization

The query optimization techniques fall into five groups namely rewriting, expansion, decomposition, pseudo document generation and iterative refinement. It depends on what they do to the query: rewriting it, expanding it, breaking it apart, generating a stand-in document for it, or refining it across multiple passes. None of these are mutually exclusive in practice; production systems often stack two or three together.

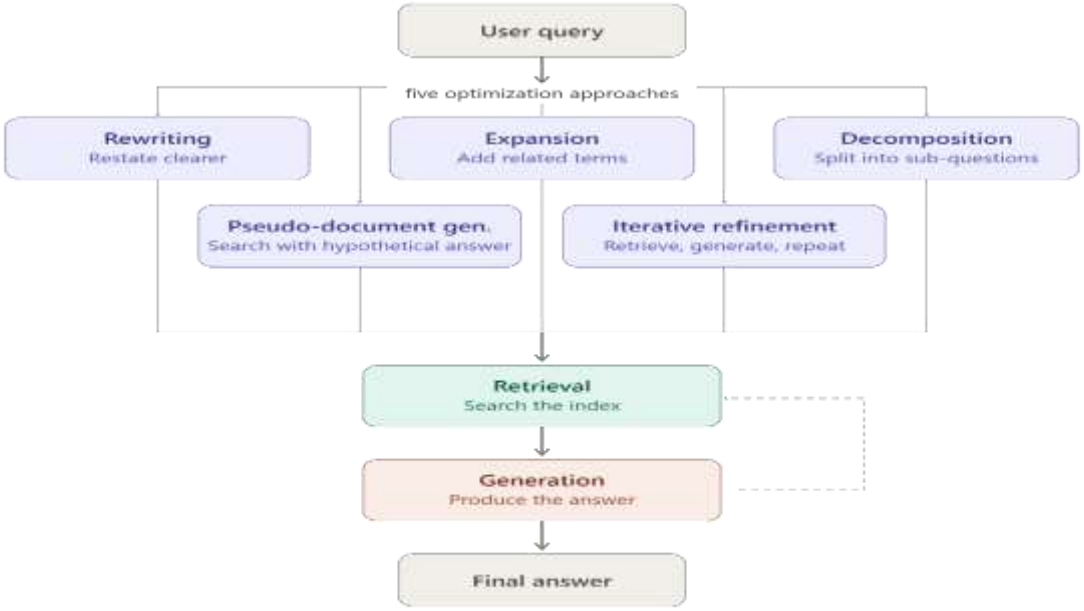


Figure 2. Optimization approach

4.1 Query Rewriting

The simplest of the bunch. The user queries and restates it in a form that's easier to retrieve against, before it ever touches the index. In a conversational setting that usually means filling in pronouns and dropped context “what about last year" turns into "what was the revenue last year”, but it can also just mean cleaning up grammar or matching the phrasing conventions of whatever corpus you're searching. Most implementations hand this off to a smaller or auxiliary LLM. That adds a little latency, but it's cheap next to the other methods here.

4.2 Query Expansion

Here you keep the original query intact but bolt on related terms, synonyms, or nearby concepts to widen the net. Older IR systems handled this with synonym lists or term co-occurrence stats; modern RAG pipelines mostly just prompt an LLM to spit out related phrasing and tack it on. It helps most with short queries that don't have much signal to begin with — though it can backfire if the added terms drag the search toward a related-but-wrong topic. That risk is worth watching for, not assuming away.

4.3 Query Decomposition

Through this process, the raw query is decomposed into structured sub-components before retrieval. For an HR-related query like "Mr. X Manager," the term "Mr. X" is mapped to the Employee Name attribute, and "Manager" is mapped to the Designation attribute. Each attribute is then assigned its corresponding value, and a separate retrieval is run for each one. The retrieved results are merged and passed to the LLM

along with the original query, allowing it to generate a precise, grounded answer: Employee Mr. X holding the Designation of Manager, rather than an incomplete or loosely related response based on the raw, ambiguous query alone.

4.4 Pseudo-Document Generation

This idea feels strange at first. Instead of searching with the question, you ask an LLM to write a fake answer first. Then you use that fake answer as the search query. The thinking behind this is simple. An answer looks more like real documents than a question does. The words and structure match better. So, the search works better too.

HyDE and Query2doc both use this method. Both show better retrieval results than just using the raw question, especially when the user's words don't match the way the documents are written. But there is a clear risk here. If the fake answer is wrong, the whole search goes in the wrong direction. This problem is worth stating clearly, not brushing aside.

4.5 Iterative and Feedback-Driven Refinement

Instead of treating retrieval as one-shot, these methods go back and forth — retrieve something, generate part of the answer, check what's missing, retrieve again. This matters most for genuinely multi-step questions where a single retrieval pass was never going to cut it. As the dashed feedback loop in the diagram shows, this is the one approach that doesn't simply feed a transformed query into retrieval once — it sits around the whole retrieve-generate cycle and decides, after generation, whether to go back for more. FLARE builds on this directly: it only triggers a new retrieval when the model's own uncertainty signals it's running out of solid ground, rather than retrieving on some fixed schedule. Koo et al. (2024) take a related approach — they score how well a candidate query aligns with the documents it pulls back, then use that score to refine the query through an auxiliary LLM, reporting around a 1.6% accuracy gain over the unrefined baseline. ERRR (Cong et al., 2024) comes at it from a different angle: before optimizing the query at all, it checks what the model already knows parametrically, so retrieval effort only gets spent on the parts of the question the model genuinely can't answer on its own.

5. A Worked Example and Comparison

A concrete case helps ground the discussion. Say a user asks an HR support assistant: "can I carry over my leave?" The query says nothing about which policy, year, or employee category applies, so a dense retriever working from it alone tends to return a scatter of loosely related HR pages rather than the specific carry-over clause.

5.1 Comparing the Techniques

Table 1 traces this query through each optimization stage. The retrieved results are illustrative rather than pulled from a live system, but they match the behavior reported in the papers cited in Section 3 for short, underspecified inputs.

Stage	Query sent to retriever	Typical retrieved result
Raw query	“Can I carry over my leave?”	Mixed HR pages; carry-over clause not in top results
After rewriting	“What is the policy on carrying over unused annual leave to the next calendar year?”	Annual leave policy found; employee-category specifics still missing

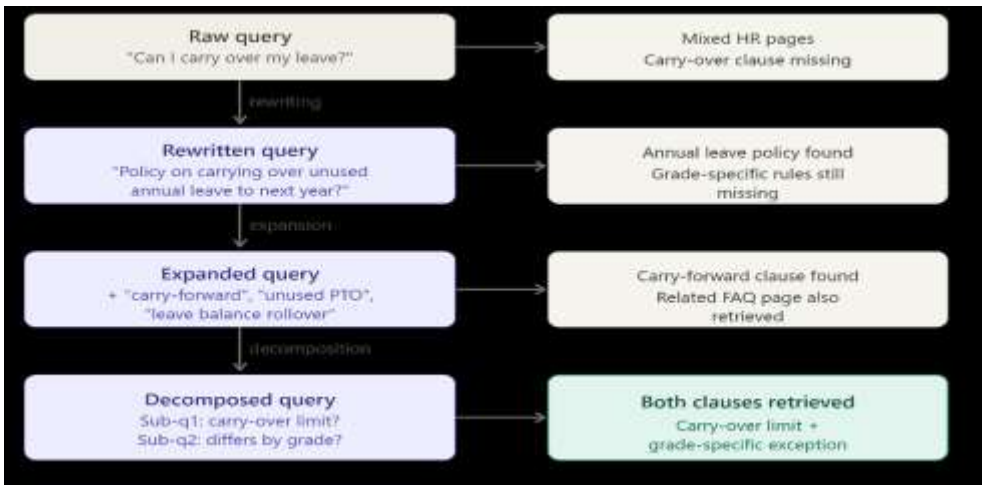
Stage	Query sent to retriever	Typical retrieved result
After expansion	Rewritten query + "carry-forward," "unused PTO," "leave balance rollover"	Carry-forward clause retrieved, plus a related FAQ
After decomposition	Sub-q1: carry-over limit? Sub-q2: does it differ by grade?	Both the limit clause and the grade-specific exception retrieved

Table 1: query reshaped

The following techniques are used for query optimization in RAG systems:

- **Rewriting**: The original query is reworded into a clearer or more specific version before it is sent to the retriever.
- **Expansion**: Extra related terms or synonyms are added to the query so the retriever can catch documents that use different wording for the same idea.
- **Decomposition**: A complex question is broken down into smaller sub-questions, and each one is retrieved separately.
- **Pseudo-doc (HyDE)**: The system first generates a hypothetical answer to the query, then uses that answer to search for real documents that match it.
- **Iterative refinement**: The query is repeatedly adjusted based on what gets retrieved, improving the search step by step until better results emerge.

Figure 3 renders the progression in Table 1 as a stage-by-stage flow rather than a static table, making explicit where each technique closes the gap between the query and the target clause and where it does not. The raw query and the rewritten query both leave a visible gap on the right-hand side of the diagram — relevant pages are retrieved, but the grade-specific exception is absent in both cases. That gap closes only at the decomposition stage, where the two sub-queries jointly retrieve both the general carry-over limit and the exception clause. This is consistent with the broader claim in Section 4.3 that decomposition is necessary specifically when a single query cannot represent more than one distinct piece of required evidence; the figure shows this as a structural property of the pipeline rather than an incidental result of this one example.



. Figure 3 Progression steps

Figure 4 plots the approximate recall gain and relative latency for all five techniques from Table 2 side by side. The two series move together almost monotonically: rewriting sits lowest on both axes, iterative refinement highest on both, with expansion, decomposition, and pseudo-document generation arranged between them. No technique in the comparison achieves a high recall gain at low latency, which is the empirical pattern underlying the trade-off discussed at the end of Section the choice of technique is therefore less a question of which method is "best" and more a question of how much added latency a given deployment can tolerate in exchange for completeness. This is the pattern that motivates the move, discussed in Section 6, toward adaptive systems that select optimization effort per query rather than applying a fixed technique uniformly

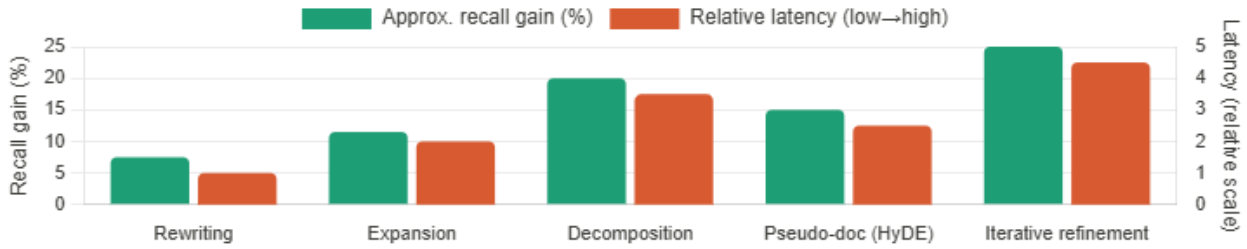


Figure 4 Plots the approximate recall gain

These techniques still fall short in a few connected ways. Almost all of them require at least one extra LLM call before generation even starts, which is fine in research but costs real milliseconds in a live chat assistant — likely the main reason adoption isn't universal, and why smaller, distilled query optimizers are gaining traction as a cheaper substitute. Measuring whether an optimized query actually helped is also murkier than scoring a generated answer: retrieval metrics like recall@k don't reliably track final answer quality, since a high-recall query can still confuse the generator with noise, and a modest-recall query can still produce a fine answer if the generator compensates, so the metrics alone often can't tell you whether a technique is genuinely helping. Expansion and pseudo-document methods add a further risk: the LLM doing the expanding or hypothesis-writing can confidently invent something plausible but wrong, and retrieval then chases that fabrication instead of the real question, a serious problem in domain like medical or legal retrieval, where confidently wrong evidence can be worse than no evidence at all. And most published gains come from general, open-domain benchmarks, so whether they hold up on a hospital's internal documentation or a law firm's case archive is genuinely unclear; adapting to a specific domain usually requires fine-tuning on domain query-document pairs or building in a domain knowledge graph, both real engineering work rather than a setting to flip on.

7. Conclusion

Query optimization isn't a side detail in RAG, it's one of the most important parts to get right, since nothing downstream fixes evidence pulled from a wrong query. The five techniques each solve a different version of this problem, and combining a couple usually works better than picking one. Recent work points toward agentic systems that judge query difficulty first, then decide how much optimization effort to spend, keeping simple queries fast while giving complex one’s heavier treatment. Self-RAG style checks on retrieval and answer quality, plus ERRR's "check what's already known first" idea, both seem underexplored. Nothing here is solved yet, but this direction beats just stacking more retrieval steps.

References

1. Cong, Y., Akash, P. S., Wang, C., & Chang, K. C. (2024). Query Optimization for Parametric Knowledge Refinement in Retrieval-Augmented Large Language Models. arXiv preprint arXiv:2411.07820.
2. Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T. S., & Li, Q. (2024). A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. arXiv preprint arXiv:2405.06211.
3. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., & Wang, H. (2023). Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv preprint arXiv:2312.10997.
4. Koo, H., Kim, M., & Hwang, S. J. (2024). Optimizing Query Generation for Enhanced Document Retrieval in RAG. arXiv preprint arXiv:2407.12325.
5. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems, 33.
6. Peng, B., Zhu, Y., Liu, Y., Bo, X., Shi, H., Hong, C., Zhang, Y., & Tang, S. (2024). Graph Retrieval-Augmented Generation: A Survey. arXiv preprint arXiv:2408.08921.
7. Yu, Z., Xiong, C., Yu, S., & Liu, Z. (2023). Augmentation-Adapted Retriever Improves Generalization of Language Models as Generic Plug-In. Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics.
8. Sudhakar Ranjan, Komal Kumar Bhatia "Query interface integrator for domain Specific Hidden web" International Journal of Computer Engineering and Applications (IJCEA) ISSN 2321-3469, Vol. IV, Issue I/III, Oct.-Dec. 2013.
9. Sudhakar Ranjan, Komal Kumar Bhatia "Indexing for Vertical Search Engine: Cost Sensitive "International Journal of Emerging Technology & Advanced Engineering (ISSN 2250-2459, ISO 9001:2008 Certified Journal), Volume 3, Issue 10, October 2013.
10. 3. Sudhakar Ranjan, Komal Kumar Bhatia"Web Log for Domain Specific Hidden Web", International Journal of Computer Engineering and Applications (IJCEA) ISSN 2321- 3469, Vol VII, Issue I, July 2014.
11. Sudhakar Ranjan, Komal Kumar Bhatia "Indexing for Domain Specific Hidden Web" International Journal of Computer Engineering and Applications (IJCEA) 2321-3469, Vol VII, Issue I, July 2014.
12. Sudhakar Ranjan, Komal Kumar Bhatia "Transaction in Hidden Web" International Journal of Computer Engineering and Applications (IJCEA) 2321-3469, Vol VIII, Issue Teaching Area/ Research Area Experience Research Credentials Experience 4 II, Nov 2014.
13. Ranjan, Sudhakar, and Komal Kumar Bhatia. "Design of a Least Cost (LC) Vertical Search Engine Based on Domain Specific Hidden Web Crawler." The Dark Web: Breakthroughs in Research and Practice, edited by Information Resources Management Association, IGI Global Scientific Publishing, 2018, pp. 319-333. https://doi.org/10.4018/978-1-5225-3163-0.ch014